www.ibpinet.net/wdesigner

Formação Profissional
WEB DESIGNER

IBPINET
The Internet School

Criação de Web Sites 2

Introdução ao servidor Web
Aplicações Web interativas
Programas CGI e SSI

CWS2-04-2000/01

Helder da Rocha

Helder da Rocha

Criação de
Web Sites II

Servidor Web
Aplicações Web
Programas CGI e SSI

Rev.: CWS2-04-2000/01 A4

Copyright © 2000 por Helder Lima Santos da Rocha. Todos os direitos reservados. Os direitos do autor sobre esta obra estão protegidos pela Lei 9.610/98 (Lei de Direitos Autorais).

Nenhuma parte desta publicação poderá ser reproduzida ou utilizada em outros cursos sem a autorização, por escrito, do autor. Alunos, professores ou funcionários de instituições educacionais ou filantrópicas podem requerer autorização gratuita para a reprodução deste material para uso próprio, ou para uso em treinamentos sem fins lucrativos. O autor pode ser contatado através dos endereços eletrônicos hlsr@uol.com.br ou helder@ibpinet.net.

Este livro faz parte da série "Criação de Web Sites" de propriedade de Helder Lima Santos da Rocha. Esta edição destina-se exclusivamente ao uso como material didático do curso "Formação WebDesigner", da IBPI Software e Serviços Ltda. e não pode ser vendida separadamente nem utilizada como material didático de outros cursos.

Embora todos os cuidados tenham sido tomados na preparação deste livro, o autor não assume responsabilidade por erros e omissões, ou por quaisquer danos resultantes do uso das informações nele contidas.

Web Site: http://www.ibpinet.net/helder/wd/cws2/

Capa, editoração eletrônica e revisão: o autor.
Código desta edição: **CWS2-03-2000/01** (Primeira edição)
Formato: A4 – livro (numeração contínua).
Responsável por esta tiragem: IBPINET São Paulo
Tiragem desta edição: 80 cópias (juntamente com a edição "0" para a IBPINET).

R672c2 Rocha, Helder Lima Santos da, 1968-

"Criação de Web sites II – tecnologias de cliente/servidor". 1a. edição com tiragem limitada e autorizada apenas para cursos da IBPINET realizados até 31/12/2000.
190 p. A4. Acompanha disquete ou CD de 3 ¼" ou *Web site* com código e exemplos.
Reprodução: IBPINET – São Paulo, SP, Abril de 2000.

1. Internet (rede de computadores, meio de comunicação) – Programação. 2. Hypertext Transfer Protocol – HTTP (Protocolo de comunicações da Web). 3. CGI – Common Gateway Interface (Mecanismo de integração para execução de aplicações remotas). I. Título: Criação de Web Sites II – Tecnologias cliente/servidor. II. Série.

CDD 004.678

Quaisquer marcas registradas citadas nesta obra pertencem aos seus respectivos proprietários.

Conteúdo

Prefácio

Parte 1: Servidor e Plataforma Web

1. *Princípios de informática para a Web*
2. *Sistemas operacionais da plataforma Web*
3. *Servidor Web*
4. *Aplicações Web*
5. *Instalação do Serviço de Aplicações*

Parte 2: Formulários, HTTP e CGI

6. *Formulários*
7. *Formulários e CGI*
8. *Princípios de HTTP*
9. *Uso de programas CGI*

Parte 3: SSI, Cookies e Programação

10. *Server-side includes*
11. *Cookies*
12. *Princípios de programação*

Apêndice A – Linguagem Perl

Introdução

1. *Sintaxe básica*
2. *Entrada e saída*
3. *Expressões regulares*
4. *Subrotinas e bibliotecas*
5. *Transformação de dados*

6. Módulos
7. Referências

Apêndice B – Bibliografia

Prefácio

Este livro contém textos selecionados especialmente para o curso *Criação de Web Sites II* com a finalidade de servir de apoio ao curso como fonte de informações teóricas e exercícios. Ele não é a única fonte de informações do curso (o instrutor poderá indicar outras) nem seu conteúdo necessariamente faz parte do curso (há material que poderá ou não ser usado, a critério do instrutor).

O curso *Criação de Web Sites I* é o segundo da série de três cursos, de igual duração, que exploram a maior parte das tecnologias essenciais e de domínio público para criar Web sites interativos. Enquanto no último curso foram desenvolvidas páginas visíveis localmente no browser – a aplicação *cliente* da plataforma Web, neste curso serão explorados os múltiplos papéis do *servidor*, principalmente aquele que permite o desenvolvimento de aplicações Web. O objetivo deste curso, portanto, é familiarizar o aluno com o servidor Web e ensiná-lo a desenvolver páginas Web interativas que funcionem como aplicações cliente-servidor.

Será abordado o funcionamento do servidor *Apache* (o mais utilizado no mundo) ou do servidor *PWS/IIS* (o mais popular em plataformas *Windows*). Serão apresentados o protocolo HTTP (essencial nas comunicações da Web) e a interface CGI (usada para permitir a execução de programas remotos). Para entender como funciona CGI, o aluno terá que desenvolver formulários em HTML e utilizar programas simples disponíveis na Internet.

No final do curso o aluno terá também uma pequena introdução à programação (com a linguagem Perl, possivelmente), que também o preparará para o curso seguinte. Serão mostrados programas simples em Perl (o aluno não precisa saber Perl) para expor os princípios do CGI (que podem depois ser aplicados pelo aluno para desenvolver CGI em outras linguagens como C, C++, VB ou Delphi se desejarem). Serão também discutidos os problemas do CGI, com demonstração de exemplos em tecnologias alternativas como servlets, JSP e ASP.

O que você já deve saber

Para ter o melhor rendimento neste curso, você deverá:

- saber HTML
- estar familiarizado com tecnologias Web (como CSS), com o browser, com URLs e com a maneira como o servidor mantém as páginas (em diretórios virtuais). Esses assuntos fazem parte de um curso como "Criação de Web Sites I" ou equivalente.

O laboratório onde este curso será ministrado em geral só contém um tipo de computador (PC, Mac ou Linux – consulte a coordenação do curso para saber qual). Usuários de outros sistemas deverão revisar as tarefas básicas do sistema disponível no laboratório antes de iniciar o curso de forma a obter o melhor rendimento em sala de aula.

No Web site dedicado a este curso (veja página de *copyright*) há links para material adicional de referência, aplicativos (para Linux, Macintosh e Windows) usados em sala de aula e alternativas, programa do curso (datas) e outras informações.

Descrição do conteúdo

Esta apostila consiste de quatro partes, uma das quais é um apêndice opcional contendo assunto que não será abordado em sala de aula (serve para quem quiser se aprofundar). A primeira parte apresenta os conceitos fundamentais para compreender o funcionamento da plataforma Web – local onde irão executar as aplicações estudadas neste curso. Mostra como instalar e configurar um servidor Web para publicar páginas e instalar aplicações.

A segunda parte volta a apresentar HTML, mas apenas o HTML, deixado de fora do último módulo, necessário para a criação de interfaces de formulários. Mostra também como construir aplicações CGI e como ligá-las aos formulários. Há também um capítulo sobre o protocolo HTTP.

A terceira parte apresenta mais duas tecnologias interativas: Server Side Includes (SSI) e Cookies. Há exemplos disponíveis mas para tirar o melhor proveito de uma tecnologia como Cookies, é importante saber programar. O último capítulo, portanto, apresenta as noções básicas de linguagens de programação. Esse capítulo é essencial não só para se compreender programas em Perl, mas como requisito prévio ao curso seguinte, centralizado na linguagem JavaScript.

A quarta e última parte contém um tutorial sobre a linguagem Perl. Para os que já sabem programar, o tutorial apresenta a sintaxe básica e vários exemplos de código que podem ser testados. Perl pode ser usada como linguagem para desenvolver aplicações CGI.

Esta apostila não contém tudo sobre aplicações Web. Ela não mostra como escrever aplicações Web na maior parte das tecnologias existentes (são muitas) como ASP, JSP, servlets, PHP, e outras. Tais tecnologias exigem conhecimento não só de programação (o que não é o objetivo principal deste treinamento) como conhecimento razoável do servidor Web (o que se aprende no curso, com CGI). Para ir além, procure as fontes de informação que serão publicadas no site do curso ou consulte o seu instrutor sobre endereços na Web onde se pode encontrar maiores informações. Aplicações nessas tecnologias poderão ser demonstradas durante o curso.

Programa do curso

O programa do curso cobre a maior parte dos assuntos discutidos nesta apostila, mas não todos. Alguns assuntos são temas de outros cursos e estão na apostila apenas para servir de referência (por exemplo, o apêndice sobre Perl e certos tópicos sobre cookies e aplicações CGI). O programa abaixo não é estático. A ordem e profundidade dos assuntos abordados em sala poderão ser modificadas pelo instrutor, visando o melhor aprendizado e o cumprimento dos objetivos do treinamento.

1. INTRODUÇÃO AO SERVIDOR WEB E HTTP (PARTE 1): como instalar, configurar e usar um servidor Web no Unix e no Windows. *Tópicos:* Princípios de informática para a Web, Sistemas operacionais da plataforma Web, Linha de Comando do Windows (MS-DOS), Unix Shell (Linha de Comando no Linux), Servidor Web, O que é um servidor Web, Configuração, Instalação e configuração básica do Apache e PWS, Aplicações Web, Instalação do Serviço de Aplicações, Programas CGI, Implantação do CGI, Alternativas ao CGI, Exercícios e testes.

2. HTML PARA FORMULÁRIOS (PARTE 2): construção da interface do usuário. *Tópicos:* Construção de formulários com HTML. Como os dados são enviados para o servidor. Elemento <FORM>, <INPUT>, Botões (TYPE=BUTTON, SUBMIT, RESET ou IMAGE), Campos de texto de uma linha (TYPE=TEXT ou PASSWORD), Campos ocultos (TYPE=HIDDEN), Chaves booleanas do tipo "botão de rádio" (TYPE=RADIO), Elemento <LABEL>, Chaves booleanas tipo "caixas de checagem" (TYPE=CHECKBOX), Upload de arquivos (TYPE=FILE), Elemento <TEXTAREA>, Elementos <SELECT> e <OPTION>, Grupos de opções: elemento <OPTGROUP>, Subgrupos de componentes <FIELDSET> e <LEGEND>, Botões HTML 4.0 <BUTTON>. Testes e exercícios.

3. FORMULÁRIOS E CGI (PARTE 2): interligando formulários HTML com aplicações CGI. *Tópicos*: Variáveis de Ambiente, Princípios de HTTP, Uso de programas CGI, Onde conseguir programas CGI,

Como configurar e instalar os programas, Como criar programas CGI simples. *Tópicos adicionais (material anexo):* Interligando formulários com aplicações alternativas a CGI: ASP, JSP, ISAPI ou servlets. Demonstração de aplicação de acesso a banco de dados Access usando CGI e ASP ou JSP e servlets Java. Discussão sobre vantagens/desvantagens. Testes e exercícios.

4. SERVER-SIDE INCLUDES, COOKIES E PROGRAMAÇÃO (PARTE 3): como usar server-side includes para incluir texto em páginas geradas *on-the-fly* pelo servidor. Como usar cookies para manter o estado entre páginas. Como programar. *Tópicos:* Server-side includes, Como habilitar SSI no servidor, Criação de cookies, Recuperação de cookies, Princípios de programação, Tipos de linguagens, Componentes de um programa, Como escrever um programa. Testes.

5. APÊNDICE: LINGUAGEM PERL: (opcional). Para demonstrações desta seção parte pode-se usar o ActiveState ActivePerl no Windows e/ou o Perl 5 do Linux via Telnet). *Observação: o objetivo deste curso não é ensinar Perl ou qualquer linguagem de programação em profundidade. A finalidade é permitir que o aluno entenda o funcionamento do CGI e para isto, é preciso usar uma linguagem (que não precisa ser Perl). Perl foi adotada para demonstrar CGI porque é a linguagem mais popular usada com CGI e é possível utilizá-la para fazer programas simples sem precisar conhecê-la totalmente. Este módulo está disponível para que alunos interessados possam se aprofundar um pouco em Perl, se desejarem.*

Testes, exercícios e projetos poderão ser propostos para a realização fora do horário de aula. Reserve pelo menos algumas horas por semana para dedicar-se a eles. Exercícios adicionais, além de testes e projetos para avaliação (obrigatórios em alguns cursos), estarão disponíveis no Web site dedicado ao curso (veja página de *copyright*).

Mídia eletrônica e atualizações

Todos os exemplos, exercícios resolvidos e soluções de alguns exercícios propostos estão ou distribuídos em um disquete ou CD que acompanha este livro ou disponíveis na Internet. O curso também requer uma infraestrutura que deverá ser instalada em sala de aula e que você poderá querer instalar em casa ou no trabalho. Os programas necessários (para Windows, Macintosh e Unix) estão no site deste curso (link para a empresa que distribui o produto). Veja na página de *copyright* (logo após a capa desta apostila) qual o endereço na Internet onde os arquivos deste curso se encontram disponíveis para *download.*

Críticas e sugestões

Os textos contidos nesta edição estão sempre sendo revisados, atualizados e ampliados periodicamente e cada vez que são utilizados em cursos. Cuidados foram tomados para garantir a apresentação dos assuntos de forma clara, didática e precisa, mas eventualmente podem escapar erros, imprecisões e trechos de pouca clareza. Sugestões, críticas e correções são sempre bem vindas e podem ser enviadas por e-mail a `hlsr@uol.com.br` ou `helder@ibpinet.net`. Sua opinião é muito importante e contribuirá para que futuras edições possam ser cada vez melhores.

Helder L. S. da Rocha
São Paulo, 1o. de abril de 2000.

Criação de Web Sites II

1 Servidor e Plataforma Web

Conteúdo

1. Princípios de informática para a Web *4*
1.1. Componentes de um computador 4
Sistema operacional 5
Arquivos e diretórios 5
Sistema de arquivos 1
Aplicações cliente-servidor 6
Plataformas 6
1.2. Programação do computador 6
Linguagens 7
Linguagens de alto nível e de baixo nível 8
Linguagens declarativas, procedurais e baseadas em objetos 9
2. Sistemas operacionais da plataforma Web *10*
2.1. Linha de Comando do Windows (MS-DOS) 10
Operações básicas 11
Navegação no sistema de arquivos 12
Criação e destruição de arquivos e diretórios 13
Listagem do conteúdo de arquivos e redirecionamento 14
Variáveis de ambiente (propriedades do sistema) 14
Aplicações de linha de comando 15
Arquivos de lote 15
Aplicações de rede 16
2.2. Unix Shell (Linha de Comando no Linux) 16
Acesso remoto via telnet 17
Organização do sistema de arquivos 17
Operações básicas 18
Navegação no sistema de arquivos 18
Permissões 20
Criação e destruição de arquivos 20
Listagem do conteúdo de arquivos e redirecionamento 22
Mudança de senha e encerramento da sessão 22
Variáveis de ambiente 23
Aplicações de linha de comando 24
Roteiros Shell 24
Aplicações de rede 25
2.3. Exercícios 26
Linha de Comando do Windows (MS-DOS) 26
Unix Shell (Linha de Comando no Linux) 27
3. Servidor Web *28*
3.1. O que é um servidor Web 28
Comunicação entre agentes Web 28
Requisição para execução de programas 29
Software de servidores Web 30
3.2. Configuração 30
Porta de serviços 30
Arquivos de índice 30
Raiz de documentos 31
Aliases ou mapeamentos 31
Diretórios executáveis 31
Tipos de arquivos 31
Controle de acesso 32
Por que configurar um servidor Web? 32

3.3. Instalação e configuração básica do Apache ... 32
Instalação ... 33
Configuração básica ... 33
Configuração: porta de serviços ... 34
Configuração: raiz de documentos ... 34
Configuração: documentos padrão ... 34
Configuração: mapeamentos ... 35
Configuração: nome do servidor ... 35
Estatísticas de erro e acesso ... 35
Como publicar páginas ... 35
Instalação de programas ... 36
Controle de acesso no Apache ... 36
Configuração: arquivo de acesso ... 39
3.4. Instalação e configuração básica de servidores Microsoft ... 39
Publicação de páginas ... 40
Instalação de programas ... 40
Controle de Acesso ... 40
3.5. Exercícios ... 40
4. Aplicações Web ... 42
4.1. O que são aplicações Web ... 42
Aplicações Web com resursos lado-cliente ... 42
Aplicações Web com recursos lado-servidor ... 43
5. Instalação do Serviço de Aplicações ... 44
5.1. Programas CGI ... 44
5.2. Implantação do CGI ... 44
Servidores Unix ... 45
Servidores Windows ... 46
5.3. Alternativas ao CGI ... 46
5.4. Exercícios ... 48
5.5. Testes ... 49

1. Princípios de informática para a Web

Hoje em dia é perfeitamente possível utilizar um computador para digitar textos, desenhar, pintar, fazer cálculos, imprimir, capturar imagens com um *scanner*, enviar e-mail, navegar na Internet e até criar e publicar páginas sem conhecer os detalhes de como o computador funciona. A maior parte das tarefas como instalação de programas e até de placas de hardware está hoje automatizada. Se algo acontecer de errado, basta ligar para o suporte ou mandar o equipamento para a manutenção.

Infelizmente a Web ainda não chegou nesse estágio. Se todos no mundo usassem computadores iguais, talvez alguém que desenvolvesse aplicações para a Web só tivesse que apertar *Ok* em algumas janelas de diálogo. É possível, hoje, desenvolver aplicações para a Web dessa maneira, mas não em qualquer computador, nem em qualquer linguagem, nem usando tecnologias abertas, independentes de fabricante.

Para atingir os objetivos propostos neste curso, será necessário que conheçamos melhor o funcionamento do servidor Web, do browser, da forma como ocorre a comunicação entre os dois e como interferir para que essa comunicação realize mais que a simples transferência de imagens e páginas. Precisamos também saber como um programa de computador é interpretado, para que possamos tirar o maior proveito dos recursos interativos proporcionados pelo *Dynamic HTML* (DHTML) e *JavaScript* mais adiante. Como a maior parte dos servidores Web rodam em sistemas *Unix* (Linux, Solaris, HP-UX, AIX, etc.) será preciso saber como utilizar esse sistema operacional para realizar as tarefas básicas. Finalmente, será preciso utilizar o sistema *Windows* através da linha de comando do MS-DOS, para que possamos testar programas localmente no laboratório.

O objetivo deste capítulo é definir alguns termos fundamentais de informática que são essenciais para compreender o funcionamento do servidor Web, do sistema *Unix* e para poder desenvolver programas. Se você se sente à vontade com os termos definidos neste capítulo, fique à vontade para pular para o capítulo seguinte. As definições são superficiais. Quem quiser saber mais, deve procurar um bom livro sobre o assunto.

1.1. Componentes de um computador

Pode-se dividir um computador em duas partes: o *hardware*, que consiste de todas as suas partes físicas incluindo o monitor, o teclado, o mouse, a memória e a CPU (unidade central de processamento); e o *software*, que é a parte imaterial do computador, consistindo do sistema operacional (*Windows*, *MacOS*, *Linux*), protocolos de rede, aplicativos, etc.

O software chegou a um nível alto o suficiente para que a maior parte dos profissionais que utilizam computadores não precisem se preocupar com os detalhes do hardware. Nos concentraremos, portanto, nos componentes principais do software.

Sistema operacional

A comunicação com o computador seria árdua e impraticável se não existisse o *sistema operacional* – programa responsável por oferecer uma interface ao usuário para que ele possa controlar os dispositivos do hardware. Através do sistema operacional é mais fácil, por exemplo, identificar um bloco de informações guardado na memória como sendo um *arquivo* ou *diretório*, e criar, mover ou apagar essas informações de forma segura.

O sistema operacional também oferece uma interface para que o usuário possa executar *aplicações* e para que o programador possa desenvolvê-las, sem precisar conhecer a maior parte dos detalhes do hardware. Para o usuário, o sistema operacional nada mais é que um programa que permite rodar outros programas e controlar dispositivos externos.

Antigamente os sistemas operacionais consistiam de um conjunto de comandos que se podia digitar no teclado de um computador para iniciar um programa ou carregar dados de uma fita ou disco. Não havia interface gráfica. Os comandos básicos eram simples mas o fato de ter que digitar comandos parecia "programação" e isto mantinha o computador distante das pessoas que tinham aversão à programação. Através da evolução do sistema operacional gráfico, o computador finalmente tornou-se acessível para a maior parte das pessoas que hoje podem usá-lo e ignorar até mesmo o que seja um "sistema operacional."

Mas quem desenvolve qualquer coisa para a Internet não pode ignorar a existência dos sistemas operacionais. Um Web designer precisa lidar com vários deles pois a Internet é formada por computadores diferentes que se comunicam com os humanos usando sistemas operacionais diferentes. A grande maioria dos servidores Web está instalada em máquinas Unix. Para controlar acesso a uma parte do site, instalar programas de busca, contadores e outros aplicativos, além de administrar o sistema de arquivos onde reside um site é preciso conhecer um mínimo sobre o sistema operacional local. No caso da Internet, isto geralmente significa saber trabalhar com sistemas de arquivos e diretórios em *Unix*.

Os sistemas operacionais mais populares têm várias semelhanças entre si, o que facilita o seu uso. Além disso, os sistemas modernos possuem interfaces gráficas baseadas em janelas que podem ser operadas através de um mouse, tornando o seu uso mais simples ainda. A desvantagem é que essa interface gráfica geralmente não é disponível ao usuário que acessa a máquina remotamente. Ele precisa saber usar o sistema operacional através de comandos através de uma *linguagem* interativa como *Bourne Shell* (*Unix*) ou MS-DOS (*Windows*). Através dessas linguagens, é possível criar diretórios (pastas), removê-los, copiar arquivos e realizar todas as tarefas que o sistema operacional permite *sem* a necessidade de ambiente gráfico.

Arquivos e diretórios

O sistema operacional gerencia a memória *persistente* (disco, CD) do computador organizando-a em entidades abstratas como *arquivos* e *diretórios*. Um arquivo forma uma unidade de informações: contém dados armazenados na memória que devem ser usadas em conjunto, mas o sistema operacional pode distribuir essas informações pelo disco para tornar a sua gravação ou recuperação mais eficiente. Um diretório (ou *pasta*, nos sistemas gráficos) é um tipo de arquivo que contém uma lista de *endereços* para o início de outros arquivos.

Sistema de arquivos

O sistema de arquivos é a organização hierárquica de arquivos e diretórios. Para o usuário, existe pouca diferença entre um sistema de arquivos *Windows* e um sistema de arquivos *Unix*. Em ambos é possível criar, copiar, remover e mover arquivos e diretórios. Ambos permitem a organização hierárquica de arquivos e diretórios. Nos sistemas *Unix*, porém, há

maior controle sobre o nível de acesso permitido a cada arquivo ou diretório.

Tanto em sistemas *Unix* como em sistemas *Windows* há arquivos especiais que não servem apenas para armazenar informações estáticas, mas dados que serão lidos como *instruções* com ordens que devem ser *executadas* pelo processador. Esses são os arquivos *executáveis*. No *Windows* eles são identificados pela sua extensão (`.exe`, `.bat`, `.com`). Já nos sistemas *Unix*, qualquer arquivo pode ser rotulado como executável, independente de sua extensão.

Quando um programa está executando, suas instruções são transferidas para a memória RAM (não persistente) do computador e processadas pela unidade central de processamento. Um programa em execução é chamado de *processo* ou *tarefa*. Os sistemas operacionais modernos podem manter vários processos ativos ao mesmo tempo, mesmo quando o processador só é capaz de interpretar uma instrução por vez. Gerenciando o tempo que cada processo utiliza para executar instruções na unidade de processamento, dá a impressão, ao usuário, que várias coisas acontecem ao mesmo tempo. Esses sistemas operacionais são chamados de sistemas operacionais *multitarefa*.

Outros sistemas permitem que vários usuários compartilhem o processador ao mesmo tempo. São sistemas operacionais *multiusuário*.

Aplicações cliente-servidor

A maior parte das aplicações de um computador são executadas pela unidade de processamento local. Há certas aplicações chamadas de aplicações *distribuídas* que são apenas parcialmente executáveis localmente. Parte delas roda em uma ou mais máquinas remotas acessíveis através da rede do qual o computador faz parte. As aplicações distribuídas mais comuns são aplicações *cliente-servidor*.

Aplicações cliente-servidor consistem de no mínimo duas partes. Um processo *servidor*, que roda continuamente esperando instruções remotas e um ou mais processos *cliente*, que podem ser temporários, e enviam instruções que devem ser atendidas pelo servidor. Na Internet, qualquer máquina que ofereça um serviço numa rede TCP/IP exerce o papel de servidor. Usamos, portanto, o termo servidor para nos referirmos tanto à *máquina* que oferece serviços, quanto ao *software* que, executando (como um processo), torna esse serviço disponível.

Plataformas

O conjunto sistema operacional mais processador recebe freqüentemente o nome de *plataforma*. A plataforma Windows-PC, por exemplo, consiste de uma máquina PC rodando Windows. Exemplos de outras plataformas são a plataforma MacOS-Macintosh, Solaris-PC, Solaris-SPARC, Linux-PC, etc. Geralmente, aplicações são desenvolvidas para uma determinada plataforma. O mesmo programa que roda em Solaris-PC geralmente não roda em Solaris-SPARC nem em Windows-PC. Uma exceção são programas desenvolvidos em Java – linguagem que foi criada com a finalidade de permitir o desenvolvimento de aplicações multiplataforma.

Uma aplicação cliente-servidor estende-se além dos limites físicos de um computador. Ela roda na rede que pode ser vista como um grande computador virtual ou plataforma. A *plataforma Web* é o meio onde rodam aplicações distribuídas que se comunicam via HTTP. Consiste do conjunto de servidores HTTP da Internet que mantém no ar os sites, que podem ser visitados pelo conjunto ainda maior dos browsers ou clientes HTTP.

1.2. Programação do computador

Para que um computador possa realizar alguma tarefa, é preciso que ele receba instruções sobre *o que fazer* numa linguagem que ele entenda. Os primeiros computadores eram programados com seqüências de chaves ligadas ou desligadas que armazenavam informação (os *dados*) e seqüências de instruções (o *programa*). Nos computadores modernos, o sistema operacional assumiu a tarefa mais árdua de se comunicar diretamente com o hardware do computador e oferecer a infraestrutura mínima para iniciar a execução de programas, o que hoje permite que as pessoas usem computadores sem precisar saber programar. A programação também tornou-se mais fácil atra-

vés do desenvolvimento de linguagens chamadas de *alto-nível*, que permitiam que o desenvolvedor expressasse *instruções* em uma linguagem parecida com a linguagem falada, que depois eram traduzidas para a *linguagem de máquina*, compreendida pelo computador.

Mas para a maior parte das tarefas hoje em dia, não é necessário saber programar. Antigamente, um desenho gerado por computador consumia anos de trabalho e era tema de teses de doutorado em computação gráfica. O desenhista tinha que necessariamente conhecer muito bem o computador e saber muita programação. Hoje, o mesmo desenho é feito por desenhistas que usam o computador como uma ferramenta de trabalho, e eles não precisam saber programar. Para desenvolver aplicações para a Web, há, semelhantemente, diversos aplicativos que geram código automaticamente. Além disso, o trabalho hoje pode ser facilmente dividido. Um Web designer pode elaborar a parte gráfica e visual de um site e contratar um programador para fazer a parte que exige programação. Por que então aprender a programar? Quais os benefícios da programação para um Web Designer?

A programação não é o objetivo principal deste curso. Acreditamos que não é necessário a um artista gráfico aprender uma linguagem como *PostScript*, *Lingo* ou *Lisp* para fazer desenhos sofisticados sem uma ferramenta de desenho, animações *Shockwave* ou modelagem em um software como o *AutoCAD*. Os que sabem, porém, têm mais poder para ir *além* do que o software oferece. Na Web esse fato é mais relevante ainda, por ela ser uma invenção recente e estar distribuída por máquinas diferentes.

A finalidade desta seção é apresentar uma breve introdução às linguagens de programação. Em um capítulo posterior será exposta uma introdução à *lógica* de programação. Com isto acreditamos que será mais fácil encarar neste módulo e principalmente no módulo seguinte a linguagem *Perl* e a linguagem *JavaScript*. A linguagem *Perl* será objeto de uma abordagem mais superficial, por ser mais complexa e de pouca utilidade para a grande maioria dos Web designers (a não ser aqueles que também são programadores). A linguagem *JavaScript*, porém, interage diretamente com o HTML e permite a construção de interfaces com alto nível de interatividade, usando relativamente poucas linhas e exigindo poucos conhecimentos de programação. É essencial para quem pretende criar sites dinâmicos, usar DHTML ou ter maior controle sobre programas embutidos em uma página (applets Java, *Flash*, plug-ins de som e vídeo).

Linguagens

A linguagem é o meio pelo qual os seres humanos podem se comunicar com um computador e utilizá-lo para armazenar informações de uma maneira organizada e para realizar operações sobre essas informações. *Dados* são armazenados em computadores através de linguagens *declarativas*, que descrevem sua estrutura. *Procedimentos* são realizados através de linguagens de *programação* que permitem escrever programas que irão dizer ao computador o que ele deve fazer com os dados.

Internamente, um computador digital mantém todas as suas informações através da manipulação de dois estados: ligado e desligado, logicamente representados respectivamente pelo 1 e pelo 0. Com esses dois dígitos apenas, ele consegue realizar qualquer tarefa para o qual for programado, realizando cálculos e armazenando resultados numéricos representados no *sistema binário*. Para armazenar o número 214, por exemplo, o computador armazena um padrão de bits ligados e desligados equivalente a 11010110. Como 10 não é potência de 2, é complicado representar todos os dígitos binários através de números *decimais* exatos. Por causa disso, o software dos computadores também utiliza os sistemas *octal* e *hexadecimal*. 214 equivale a 326 em octal e a D6 em hexadecimal (que utiliza um alfabeto de 16 dígitos). Cada quatro dígitos binários corresponde a um dígito hexadecimal ($2^4 = 16$). Cada três dígitos binários corresponde a três dígitos octais ($2^3 = 16$).

As informações armazenadas nos computadores podem ser interpretadas de duas formas: como *dados* ou como *instruções*. Todas são armazenadas como 0s e 1s. Suponha, como exemplo, um computador simples com apenas duas instruções: *somar* e *subtrair*. Esses comandos teriam que ser codificados em 0s e 1s. Suponha que o número 01 signifique o comando *somar* e 00 o comando *subtrair* e que o computador espere sempre por *uma* instrução e *dois* dados, de forma alternada. O seguinte programa: 010101 somaria 01 e 01:

`01` Este é o *comando* "soma". O computador espera agora duas linhas de dados.

`01` Este é o primeiro argumento da soma. Falta um.

`01` Este é o último argumento. O computador agora espera um novo comando.

Observe que o primeiro 01 foi interpretado como instrução, enquanto que os dois seguintes foram considerados informação para ser alimentada ao programa. Os computadores modernos têm bem mais instruções e operam com números binários de até 64 dígitos, que são usados para representar instruções elementares, endereços de memória, cores de pixels de tela, posições de pixels na tela, caracteres, eventos do mouse, sinais do mouse, da rede, etc.

Linguagens de alto nível e de baixo nível

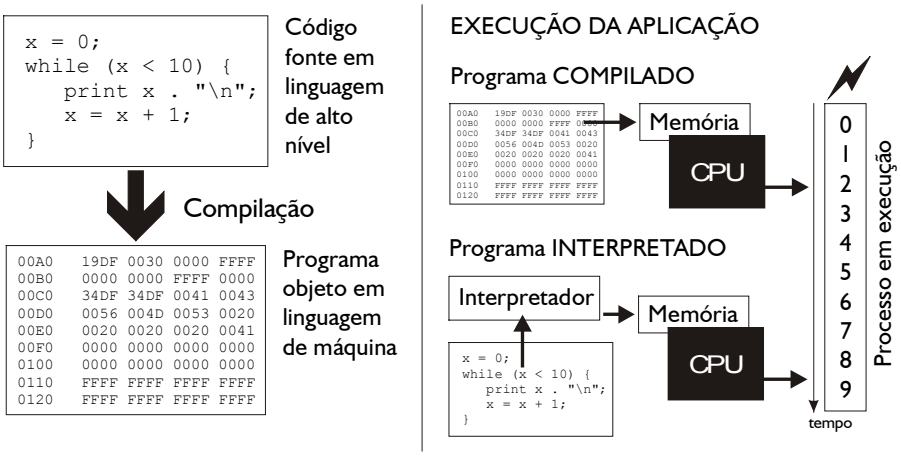
Programas escritos usando apenas 0s e 1s são programas escritos em *linguagem de máquina*. Fazer uma aplicação gráfica com um computador que só sabe manipular 0s e 1s exigiria milhares ou milhões de linhas de código. Escrever ou até analisar tais programas é uma tarefa árdua e na maior parte dos casos, desnecessária hoje em dia, com as modernas linguagens de programação chamadas de *alto nível*. As linguagens de alto nível precisam ser traduzidas para a linguagem de máquina antes que o computador possa executar os programas que forem escritos com ela.

Existem ainda linguagens de baixo nível que representam os 0s e 1s de maneira que se possa programar usando um teclado. Essas linguagens possuem um mapeamento direto com a linguagem de máquina e são chamadas de linguagens de montagem ou *assembly*. A conversão de uma linguagem de montagem em linguagem de máquina é direta e a mais eficiente possível. As linguagens de alto nível, por estarem mais distantes do computador, precisam ser traduzidas.

A tradução de uma linguagem de alto nível para uma linguagem de máquina pode ser realizada de duas formas diferentes:

- *Interpretação*: tradução da linguagem de alto nível (código-fonte) em linguagem de máquina durante a execução do programa.
- *Compilação*: conversão do programa (código-fonte) em outro arquivo executável (programa-objeto), contendo instruções otimizadas em linguagem de máquina (ou quase).

O diagrama abaixo ilustra (de forma simplificada) as diferenças entre programas compilados e programas interpretados.



A maior parte dos aplicativos que você usa no seu computador foram escritos em uma linguagem de alto nível como C++ e depois compilados. Páginas HTML e programas *JavaScript* são exemplos de códigos interpretados. *Applets Java* são programas parcialmente compilados, que ainda passam por um processo de interpretação na hora da execução.

Linguagens declarativas, procedurais e baseadas em objetos

Quando geralmente falamos de *programação* nos referimos à programação de computadores usando linguagens de propósito geral que podem ser utilizadas para construir roteiros de instruções ou *procedimentos*. Essas linguagens são chamadas de *procedurais*. Através delas é possível construir programas independentes, que têm um início e um fim, uma entrada e uma saída, e que serão capazes e realizar *operações* sobre as informações que receberem como entrada.

Linguagens como HTML são declarativas e não possuem as estruturas necessárias para construir procedimentos. Com HTML é possível descrever, de maneira estruturada, o texto de uma página, mas não há meios de fazer simples cálculos aritméticos em uma página. Para que um *'programa' HTML* tenha alguma utilidade, é preciso que ele seja processado por um outro programa. Semelhantes são as linguagens usadas para descrever imagens (GIF e JPEG). As estruturas construídas com tais linguagens raramente são chamadas de programas.

As linguagens mais modernas não são mais meramente procedurais. Elas combinam as estruturas construídas através de declarações com procedimentos definidos para operar sobre elas. São chamadas linguagens *baseadas* ou *orientadas a objetos*. *JavaScript*, por exemplo, quando embutida numa página HTML, representa os parágrafos, formulários, botões, etc. como *objetos* capazes de realizar diversas operações como reagir a um clique do mouse ou mudar de cor. É uma linguagem baseada em objetos. Programar em tais linguagens é geralmente mais fácil pois boa parte do código fica embutido nos objetos e o programador precisa escrever e entender menos código. Com as linguagens orientadas e baseadas em objetos hoje é possível desenvolver programas desenhando e arrastando componentes gráficos sem quase escrever uma linha sequer de código.

Apesar de mais simples hoje, a programação *é* uma tarefa complexa e exige experiência e dedicação. Aprender a usar uma linguagem de programação é o primeiro passo. Depois que se aprende a programar, é mais fácil aprender outra linguagem, pois as estruturas fundamentais se repetem e as principais linguagens usadas hoje têm uma sintaxe semelhante. Voltaremos a esse tema mais adiante.

Este capítulo procurou apresentar uma introdução superficial aos componentes de um computador, do sistema operacional e às linguagens de programação. O assunto deste capítulo não faz parte do programa do curso. O seu único objetivo é oferecer um *background* mínimo que poderá ser útil no decorrer do curso. Se você tiver interesse em se aprofundar em algum dos assuntos abordados aqui, deve procurar fontes específicas sobre eles.

2. Sistemas operacionais da plataforma Web

A plataforma Web é o grande computador virtual formado pelos servidores e clientes que interagem através do protocolo HTTP. Na plataforma Web, a maior parte dos computadores envolvidos utiliza um sistema operacional baseado em *Unix* ou *Windows*, sendo o *Unix* o principal sistema usado para servidores.

Se você desenvolve páginas para a Internet, há uma grande chance que a máquina que armazenará o seu site rode um sistema operacional semelhante ao *Unix*, como o *Linux*, o *Solaris* ou *AIX*. Na maior parte das vezes, o acesso à máquina que hospedará o site será feito de forma remota, e nesses casos, a sua interface com o sistema operacional geralmente será orientada a caractere, através de linha de comando.

Conhecer os comandos básicos para criar e remover diretórios e arquivos, mudar permissões e senhas é essencial para Web designers que pretendem instalar, configurar ou desenvolver aplicações Web que rodarão nas máquinas servidoras. Neste capítulo, será apresentada uma pequena introdução ao uso de sistemas operacionais através de sua interface de *linha de comando*. Veremos os comandos básicos do *Unix* e do *MS-DOS* – a linha de comando do *Windows*, que será utilizada no teste e execução de aplicações Web no laboratório.

Na seção a seguir apresentaremos os comandos básicos do MS-DOS porque ele é um dos sistemas que usaremos em laboratório. O sistema de arquivos do MS-DOS e os comandos para controlá-lo são muito semelhantes aos usados no ambiente *Unix*. Se você já conhece e sabe usar o ambiente MS-DOS fique à vontade para pular a seção seguinte.

2.1. Linha de Comando do Windows (MS-DOS)

O *Windows* é um sistema operacional gráfico *orientado a eventos*, ou seja, ele espera que o usuário ou o computador provoque *eventos* como mover o mouse, digitar uma tecla, ligar a impressora, conectar-se à Internet, etc. para que possa realizar alguma coisa. Antigamente não era assim. Antes, o *Windows* rodava sob o sistema MS-DOS (*Microsoft Disk Operating System*), que oferecia uma interface de controle através da linha de comando. Ainda é possível controlar a maior parte do sistema de arquivos e diretórios do *Windows* usando o MS-DOS e vários programas ainda precisam desse ambiente para executar.

A linha de comando é indicada por um cursor chamado de *prompt* que aguarda que o usuário digite um comando e depois o submeta ao interpretador da linha de comando, digitando a tecla *Return* ou *Enter*. Para rodar a linha de comando no Win-

dows, é preciso abrir o aplicativo *Prompt do MS-DOS* (figura ao lado).

Cada comando digitado e enviado é interpretado por um aplicativo chamado `COMMAND.COM`, localizado no diretório raiz do disco `C:`. Várias aplicações que rodam a partir do servidor Web rodam sob o ambiente MS-DOS. Para testar essas aplicações, teremos que usar comandos do MS-DOS.

Quando você roda o *Prompt do MS-DOS*, ele é inicializado com alguns parâmetros iniciais. As configurações iniciais da linha de comando MS-DOS podem ser definidas no programa `AUTOEXEC.BAT` que é executado antes do início do Windows. Esse arquivo é utilizado geralmente para definir variáveis de ambiente e rodar determinadas aplicações. Uma das aplicações úteis que você pode acrescentar no seu `AUTOEXEC.BAT` (para que ele a execute antes de abrir a janela do MS-DOS) é o `DOSKEY`. Esse programa faz com que os comandos digitados sejam lembrados e possam ser repetidos rapidamente. Para acrescentar essa instrução no `AUTOEXEC.BAT`, abra-o (ele está em `C:\`) no bloco de notas e na última linha digite:

```
doskey
```

Agora salve o arquivo, feche-o e reinicialize o *Windows*.

Operações básicas

Esta seção relaciona as operações básicas que podem ser realizadas através do sistema MS-DOS. Se você nunca usou MS-DOS, aproveite e repita os exemplos abaixo. O aplicativo *Prompt do MS-DOS* pode ser iniciado através do menu *Iniciar* do *Windows*. Ele pode ocupar toda a tela ou apenas uma janela do Windows. Para alternar entre esses dois modos de exibição digite a combinação de teclas *Alt - Enter*. No modo janela você pode ainda alterar o tamanho da fonte de letra, caso o texto esteja difícil de ler.

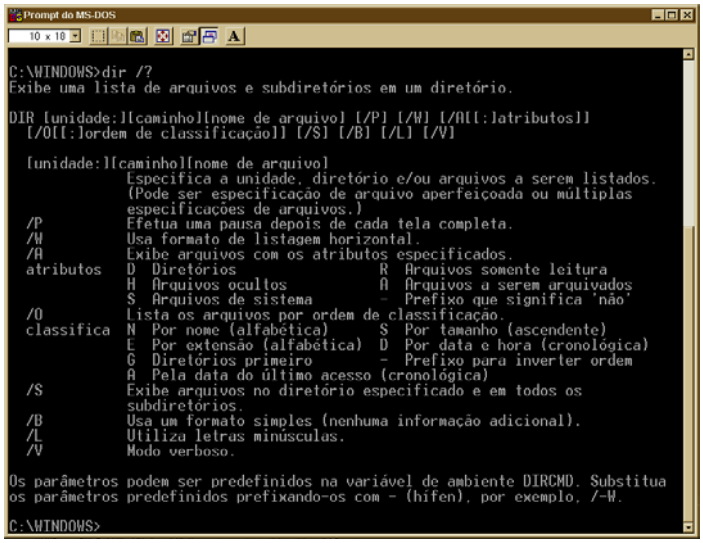
Quando você abrir a janela do DOS, o cursor da linha de comando piscará ao lado do símbolo `C:\>` (que informa o nome do *drive* de disco atual e pode ou não conter outras informações).

A sintaxe da maior parte dos comandos DOS é simples. Alguns só exigem que você digite o nome do comando, por exemplo:

```
C:\> dir
```

é o suficiente para listar o conteúdo do diretório atual.

Vários comandos do DOS possuem opções que modificam os seus resultados. A maior parte das opções consiste de argumentos de linha de comando que são precedidos pelo caractere "/" (barra). A opção `/?`, quando disponível, mostra ajuda sobre o comando. Para saber quais são os argumentos disponíveis de um comando, pode-se digitar:




```
comando /?
```

Por exemplo:

```
C:\> dir /?
```

não lista os arquivos do diretório mas as opções que podem ser usadas com `dir` (figura ao lado).

Além das opções (geralmente opcionais), alguns comandos exigem a presença de um ou mais nomes de arquivos ou diretórios como argumentos que devem aparecer depois do comando, por exemplo:

```
C:\> rename arquivo.txt novo.txt
```

O comando acima não funciona se os dois argumentos não estiverem presentes.

Todas as alterações feitas no ambiente MS-DOS repercutem no *Windows*. Se você criar um diretório no DOS, aparece uma pasta no *Windows*, se você mudar o seu nome, o nome muda no *Windows* e assim por diante. O MS-DOS é apenas outra forma de controlar o sistema.

Navegação no sistema de arquivos

Para listar todos os arquivos de um diretório, use o comando **`dir`**. Esse comando também aceita argumentos com filtros para listar apenas uma parte dos arquivos. No exemplo abaixo, listará apenas os arquivos que contém a extensão `.html`:

```
C:\> dir *.html
```

O asterisco (`*`) representa qualquer quantidade de caracteres. Se você desejar representar apenas um caractere qualquer, use a interrogação (`?`):

```
C:\> dir *.ba?
```

A instrução acima lista arquivos com extensão `.bat`, `.bak`, `.bam`, etc. Os filtros não são restritos ao comando `dir`. Podem ser usados sempre que nomes de arquivo forem usados como argumentos.

A listagem obtida através do comando `dir` mostra várias informações sobre o arquivo como hora da última alteração, tamanho e nomes. Arquivos que são diretórios aparecem com a indicação `<DIR>` do seu lado (em negrito):

```
.              <DIR>         26/11/99  18:10 .
..             <DIR>         26/11/99  18:10 ..
PT             <DIR>         26/11/99  18:11 pt
IMAGENS        <DIR>         26/11/99  18:11 imagens
INDEX~1  HTM        12.219   26/11/99  20:36 index.html
VOTE     GIF         3.959   23/09/99  15:40 vote.gif
CAPALIV  GIF         2.121   24/11/99  22:13 capaliv.gif
IBEST_~1 GIF         3.816   01/10/99   1:33 ibest_ball_5.gif
TOMO_INF GIF         4.242   24/11/99  22:14 tomo_inf.gif
TOMO_I~1 GIF         4.509   24/11/99  22:15 tomo_inf_b.gif
CAPA     CSS         1.129   15/09/99  10:39 capa.css
         7 arquivo(s)             32.995  bytes
         4 diretório(s)   11.264.224.768  bytes livres
```

Cada arquivo tem dois nomes no ambiente *Windows*. Ao listar o conteúdo de um diretório, os dois nomes são listados. O que aparece na primeira coluna, em letras maiúsculas, é o *nome MS-DOS* que possui mo máximo 8 caracteres com extensão (após o ponto) de até 3 caracteres. O que aparece na última, com caixa-mista, é o *nome do Windows*, que pode ter qualquer número de caracteres, além de espaços (ilegais em DOS). Em todas as aplicações de linha de comando que usaremos neste curso devemos sempre usar o nome *Windows*. Se o nome tiver espaços, porém, será preciso escrevê-lo entre aspas:

```
C:\> dir "Página Número 1.txt"
```

Qualquer diretório novo sempre tem dois arquivos: "**`.`**" que é um ponteiro explícito para o diretório atual, e "**`..`**" que aponta para o diretório anterior. Os comandos **`cd`** ou **`chdir`** servem para mudar de diretório. Para subir na árvore de diretórios (se você não estiver na raiz), use:

```
C:\Windows\> cd ..
```

Para mudar para outro diretório, use `cd` (ou `chdir`) com o nome correspondente:

```
C:\> cd system
D:\> cd c:\windows\system\
```

Se a mudança de diretório via `cd` for feita a partir de outro disco, o *prompt* permanecerá no disco atual. Para mudar de disco, basta digitar a letra correspondente seguida de dois pontos:

```
C:\> d:
```

Criação e destruição de arquivos e diretórios

Os comandos **`md`** ou **`mkdir`** (*make directory*) servem para criar um novo diretório abaixo do diretório atual.

```
C:\> mkdir temporario
C:\> cd temporario
C:\temporario\> md subdir1
C:\temporario\> md subdir2
```

Os comandos acima criam a seguinte estrutura de diretórios:

```
temporario\
        |___ subdir1\
        |___ subdir2\
```

Um diretório *vazio* sempre pode ser removido usando **`rd`** ou **`rmdir`**. Se ele contiver arquivos ou subdiretórios não será possível removê-lo. É preciso apagar todos os arquivos e remover todos os seus diretórios antes que se possa removê-lo. Arquivos comuns podem ser apagados usando **`del`**:

```
C:\> cd temporario\subdir1              (entra em temporario\subdir1)
C:\temporario\subdir1\> del *.*         (apaga todos os arquivos)
C:\temporario\subdir1\> cd ..           (sobre na árvore de diretórios)
C:\temporario\> rd subdir1              (remove temporario\subdir1)
```

Há uma maneira mais fácil, porém, de eliminar uma árvore inteira de arquivos e diretórios com o comando **`deltree`**, que é devastador. Ele apaga árvores inteiras de diretórios e todos os arquivos que contém:

```
C:\> deltree temporario
```

Os comandos para mover, copiar e mudar o nome são **`move`**, **`copy`** e **`rename`** ou **`ren`**. Arquivos podem ser movidos de lugar usando `move`:

```
C:\> move .\local.txt d:\arquivos\remoto.txt
```

O comando move também serve para mudar o nome de um arquivo. Para essa finalidade também existem os comandos `ren` e `rename`. As três instruções abaixo produzem o mesmo resultado:

```
C:\> move importante.txt inutil.txt
C:\> ren importante.txt inutil.txt
C:\> rename importante.txt inutil.txt
```

O comando `copy` é usado para copiar um arquivo de um lugar para outro (mantendo o original intacto). Se `copy` tiver um só argumento, esse argumento deve ser o arquivo, diretório ou *drive* que se quer copiar. Quan-

do há um só argumento o destino é o diretório atual "`.`". Quando há dois argumentos, o primeiro é a origem e o segundo o destino. A instrução:

```
C:\temp\> copy .\*.* d:\
```

copia todos os arquivos do diretório atual para o diretório raiz do disco `d:`. O uso do diretório "`.`" geralmente é opcional em MS-DOS, a não ser que haja alguma ambigüidade.

Listagem do conteúdo de arquivos e redirecionamento

O comando **`type`** lista rapidamente o conteúdo de arquivos. Se o arquivo for muito longo, pode-se usar um *pipe*, sinal "`|`", para redirecionar a saída através do programa **`more`**, que mostra uma página de informação de cada vez:

```
C:\temp\> type listagrande.txt | more
```

O redirecionamento também pode ser utilizado para criar arquivos, ler o conteúdo de arquivos e acrescentar informação em arquivos existentes usando os símbolos **`>`**, **`<`** e **`>>`**:

```
C:\> programa.exe > resultados.txt
C:\> filtro < dados.txt > resultados.txt
C:\> adiciona >> controle.txt
```

O MS-DOS também possui alguns arquivos especiais que são ligados a dispositivos como a impressora (`prn:`), o disco(`c:`, `d:`, `e:`), o teclado e a tela (`con:`). Você pode usar o dispositivo `con:`, por exemplo, para redirecionar qualquer coisa para a tela (saída padrão):

```
C:\temp\> copy lixo.txt con:
```

A instrução acima faz o mesmo que

```
C:\temp\> type lixo.txt
```

O dispositivo `con:` também representa o teclado (entrada padrão), se for colocado como primeiro argumento do `copy`:

```
C:\temp\> copy con: texto.txt
```

O comando acima copiará tudo o que o usuário digitar na tela para o arquivo `texto.txt` até que o usuário digite a combinação de teclas `Ctrl-Z`.

Variáveis de ambiente (propriedades do sistema)

Às vezes é preciso recorrer a uma determinada informação várias vezes. Para não precisar digitar tudo cada vez que a informação for requisitada, pode-se definir em DOS uma *variável de ambiente*. Ela irá durar enquanto a sessão (janela do MS-DOS) estiver ativa e poderá ser chamada quantas vezes for necessário para recuperar o valor guardado. Por exemplo, para armazenar um comando que será repetido várias vezes, pode-se fazer:

```
C:\> set COMANDO=dir *.gif
```

Para usar a variável de ambiente, ela deve ser chamada entre **`%`** e **`%`**:

```
C:\> %COMANDO%
```

A instrução **`echo`** imprime o conteúdo da variável ou o texto que recebe como argumento e pode ser usada para se ler o conteúdo de variáveis de ambiente:

```
C:\> echo O comando guardado é "%COMANDO%"
```

O comando acima imprimirá:

```
O comando guardado é "dir *.gif"
```

Você pode listar todas as variáveis de ambiente definidas para a sessão do MS-DOS usando a instrução `set` sem argumentos:

```
C:\> set

TMP=C:\WINDOWS\TEMP
TEMP=C:\WINDOWS\TEMP
PROMPT=$p$g
winbootdir=C:\WINDOWS
COMSPEC=C:\COMMAND.COM
CLASSPATH=;i:\jsdk2.0\lib\jsdk.jar;I:\CLASSES;I:\JAD\APPS;.
DJGPP=f:\gnuc\djgpp.env
PATH=F:\PERL\BIN;C:\WINDOWS;C:\WINDOWS\COMMAND;.;C:\MSQL;
BLASTER=A220 I5 D1 H5 P330 T6 E620
CMDLINE=move deposito.txt x.txt
COMANDO=dir *.gif
```

A variável `PATH` (acima em negrito) é geralmente definida no `AUTOEXEC.BAT` para informar ao sistema onde ele deve procurar pelos programas que são digitados na linha de comando. A maior parte dos comandos do MS-DOS são aplicações localizadas no diretório `c:\windows\command\` . Jamais é preciso informar todo o caminho até a aplicação pois o caminho está no `PATH`. Quando isso ocorre, basta digitar o nome do programa que o sistema o encontra. É possível alterar o `PATH` usando o comando `set`, mas se você não fizer isto no `AUTOEXEC.BAT`, a mudança valerá apenas para a sessão atual do MS-DOS.

O Windows não distingue maiúsculas de minúsculas. Tanto faz usar `SET` ou `set`, `dir a.txt` ou `DIR A.TXT`. Isto vale principalmente para o MS-DOS.

Aplicações de linha de comando

As aplicações que rodam na linha de comando são como os comandos nativos do DOS. Para executá-las, basta digitar o nome, sem a extensão opcional, quaisquer parâmetros necessários e digitar *Enter*. Aplicações que rodam em MS-DOS também podem ser iniciadas através de ícones do *Windows*. Quando essas aplicações iniciam, geralmente abrem uma janela do MS-DOS.

O contrário também ocorre. É possível digitar o nome de aplicações *Windows* (nem todas) em uma janela MS-DOS que causarão a abertura de uma janela do *Windows*. Tais aplicações não rodarão quando o MS-DOS estiver ativado fora do ambiente *Windows* ou remotamente. Só funcionam através da aplicação *Prompt do MS-DOS*.

Arquivos de lote

Se você tiver uma seqüência de instruções para realizar em DOS, pode criar um programa chamado *arquivo de lote*. Esse programa é um simples arquivo de textos com extensão `.BAT` contendo comandos MS-DOS que devem ser executados em seqüência. Para criá-lo, abra um editor de textos (como o `Edit.exe` do DOS ou o *Bloco de Notas* do *Windows*), digite os comandos MS-DOS, declarações de variáveis e/ou parâmetros desejados, salve o arquivo e execute-o simplesmente digitando o seu nome (sem a extensão) na linha de comando DOS. Há vários comandos DOS, como estruturas de controle de fluxo (não mencionadas aqui) que só fazem sentido quando usados dentro de arquivos de lote.

Suponha que você precise criar um conjunto de diretórios com nomes diferentes mas contendo os mesmos subdiretórios e arquivos. Você poderia repetir a seqüência de comandos:

```
mkdir jamaica
cd jamaica
mkdir imagens
mkdir paginas
echo. > index.html
```

A instrução `echo.` (`echo` seguido de ponto) imprime uma linha em branco. Redirecionar uma linha em branco para um arquivo é criar um arquivo vazio.

Criando um arquivo de lote com os comandos acima, poderíamos digitar uma única instrução, da forma:

```
C:\temp\> criadir haiti
C:\temp\> criadir guatemala
```

e ter cada uma delas repetir todos os comandos necessários para criar os diretórios e arquivos desejados. Para isto, basta colocar os comandos dentro de um arquivo de texto e salvá-lo com o nome `criadir.bat`. Para receber parâmetros de linha de comando (o nome `haiti`, `guatemala`, etc) um programa em lote pode usar as variáveis `%1`, `%2`, etc. `%1` corresponde ao primeiro argumento após o nome do arquivo, `%2` ao segundo e assim por diante. O conteúdo do arquivo, portanto, pode ser:

```
@echo off
mkdir %1
cd %1
mkdir imagens
mkdir paginas
echo. > index.html
cd ..
```

A primeira linha do programa acima desliga o eco local (para que não se veja a chamada de cada comando). A última linha volta para o diretório onde estava o programa. Ao se executar:

```
C:\temp\> criadir japao
```

A variável `%1` será substituída pelo valor `japao` e será criado um diretório `japao` contendo os subdiretórios `paginas`, `imagens` e um arquivo `index.html` vazio.

O arquivo `AUTOEXEC.BAT` é um arquivo de lote que é chamado automaticamente pelo sistema *Windows* para definir os parâmetros de inicialização do ambiente MS-DOS e rodar comandos e aplicações.

Aplicações de rede

Existem também aplicações de linha de comando que funcionam em rede, requerendo uma conexão ativa para funcionarem perfeitamente. No *Windows* nativo existe um cliente *FTP* orientado a caractere (controlado por comandos) e um cliente *Telnet* – programa que permite a conexão remota a um outro computador (através de uma interface orientada a caractere) de uma rede TCP/IP. A máquina remota deve ter um servidor *Telnet* rodando na porta 23 (*default*) ou em uma porta conhecida pelo cliente.

2.2. Unix Shell (Linha de Comando no Linux)

O *Unix Shell* é a linha de comando do *Unix*. Diferentemente do MS-DOS o *Shell* não é nada limitado. É uma linguagem de programação que dá acesso completo e absoluto ao sistema operacional. Os programas de lote *Shell* são muito mais poderosos e podem ser usados para controlar todo o computador e toda uma rede.

Neste capítulo temos misturado freqüentemente os nomes *Unix* e *Linux*. Eles não são a mesma coisa embora sejam parecidos. O *Unix* é um sistema antigo de praticamente três décadas de idade, usado em vários com-

putadores de grande porte. *Linux* é uma moderna variação do *Unix*, que se tornou o principal sistema operacional usado nos servidores Web no mundo. As diferenças são pequenas. Os comandos que veremos a seguir são universais, funcionam em qualquer variação do *Unix* como o *Linux*, o *SGI*, o *Irix*, o *HP-UX*, o *Solaris* e o *AIX*.

Acesso remoto via telnet

O *Unix* possui vários programas nativos para a comunicação em rede. Um deles é o *Telnet*, que permite o acesso remoto a um outro computador de uma rede TCP/IP. No *Windows* apenas o *cliente Telnet* estava disponível. Nos sistemas *Unix*, geralmente o *servidor* roda como um processo ativo, permitindo que outras máquinas, *Unix* ou não, tenham acesso remoto (desde que não haja restrições de segurança).

Para repetir os exemplos desta seção, utilizaremos o *Telnet* do *Windows* para ter acesso a uma máquina *Unix* acessível através da rede do IBPINET. Esse serviço só está disponível em uma determinada porta e para máquinas cliente pertencentes ao mesmo domínio.

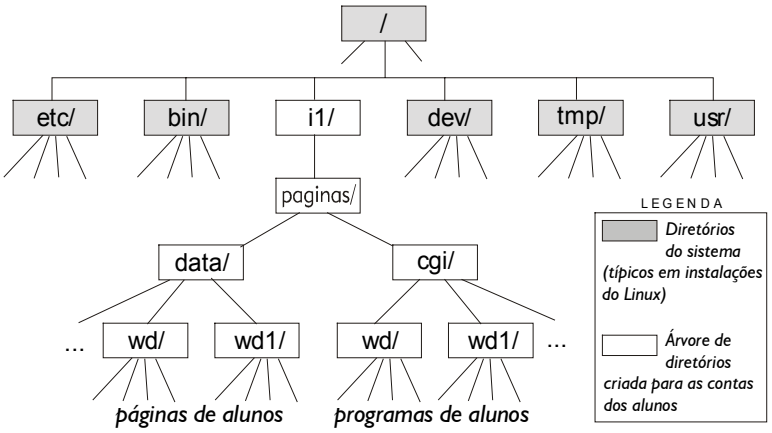
Unix é um sistema *multiusuário* (que permite que vários usuários o utilizem ao mesmo tempo) e *multitarefa* (suporta múltiplos processos rodando simultaneamente). Sistemas multiusuário geralmente têm um mecanismo de autenticação para identificar usuários através de nomes e senhas e restringir seus níveis de acesso. Para ter acesso à máquina *Unix* do IBPINET, você precisa ter uma conta no servidor pois o sistema solicitará seu nome e senha na hora que for entrar no sistema.

Organização do sistema de arquivos

Diferentemente do DOS, que possui um sistema de arquivos para cada disco, o sistema de arquivos do *Unix* começa em uma única raiz chamada de `/`. Abaixo dela estão todos os arquivos que exercem o papel de diretórios, dispositivos (discos, impressoras, redes, etc.), vínculos e outros arquivos especiais.

Tudo em *Unix* é representado através de arquivos. Cada arquivo possui um usuário *dono* e cada usuário pertence a um determinado *grupo*. Para todos os arquivos do sistema é possível definir permissões de *execução*, *leitura* e *alteração* que se aplicam ao usuário que o possui, ao grupo de usuários e ao restante dos usuários do sistema.

Quando você se conecta a um sistema *Unix* através de *Telnet* e entra com o seu nome e senha, você cai em seu *diretório casa*, onde, dentro de limites estabelecidos pelo administrador do sistema, você pode criar arquivos e diretórios, movê-los, apagá-los e executá-los à vontade. Você também pode ter acesso a arquivos que estão fora do seu diretório casa, mas em geral o acesso é restrito à leitura ou execução.



Abaixo do diretório raiz (/) está todo o sistema de arquivos do *Unix*. Há vários diretórios usados pelo sistema como `/bin`, `/dev`, `/etc`, `/lib`, `/usr` e `/tmp`. Eles contém dispositivos como discos e impressoras (`/dev`), programas compartilhados pelos usuários (`/usr`), arquivos temporários (`/tmp`), informações sobre arquivos e usuários (`/etc`), programas (`/bin`) como o interpretador de linha de comando (*Shell*) e outros dados importantes para o sistema. Os nomes desses diretórios variam entre diferentes sistemas *Unix*. A maioria pertence ao `root`, que é o nome do administrador do sistema – o "ser" onipotente e onipresente que rege o sistema *Unix*. A figura acima ilustra parte do sistema de arquivos da máquina *Linux* usada no laboratório.

Operações básicas

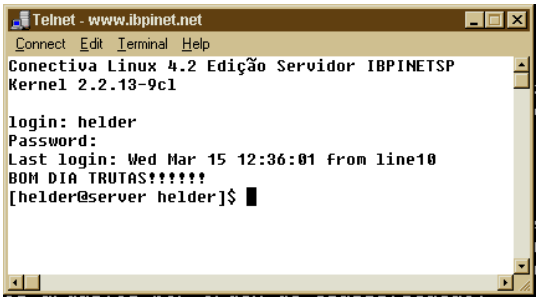
Esta seção listará os comandos básicos que você precisa para poder gerenciar um Web site armazenado em um sistema *Unix*. Se você não conhece o sistema *Unix*, aproveite para rodar todos os exemplos abaixo.

No laboratório, o acesso ao sistema *Linux* do provedor será realizado através de *Telnet*. Por razões de segurança, o sistema *Linux* da sua conta no IBPINET só é acessível via *Telnet* se você estiver dentro do domínio `ibpinet.net`. O servidor *Telnet* do IBPINET não funciona na porta *default*. É preciso especificar o número a porta após o nome ou endereço do servidor. Para rodar o *Telnet*, digite:

```
telnet www.ibpinet.net 34577
```

na linha de comando do MS-DOS ou através da opção *Executar...* do menu *Iniciar* do *Windows*. O número da porta poderá não ser o mesmo número acima (a porta de acesso local é alterada periodicamente).

Depois que você conseguir se conectar ao servidor, digitar seu nome e senha ao ser solicitado, deverá surgir na tela o símbolo `$`, que representa a linha de comandos do *Linux* (veja figura). Você então pode digitar os comandos que desejar seguido de *Enter* para enviá-los para execução.



Navegação no sistema de arquivos

Após digitar nome e senha, você deve ter "aterrissado" no seu diretório casa. Para saber qual a localização do seu diretório atual no sistema *Linux*, use o comando **`pwd`**. Os comandos que você deve digitar aparecem em negrito logo após o *prompt* do sistema (`$`). O resultado aparece em fonte *Courier* normal.

```
$ pwd
/i1/paginas/data/wd/dracula
$ _
```

Você pode agora navegar em todo o sistema de arquivos e diretórios por onde você tiver permissões de leitura e entrada usando **`cd`** ou **`chdir`** da mesma maneira que em MS-DOS. Para voltar para o seu diretório casa é só digitar **`cd`**, sem argumentos. Veja um exemplo:

```
$ cd ..          (sobe um nível na árvore de diretórios)
$ pwd
/i1/paginas/data/wd
$ cd /           (sobe até a raiz)
$ pwd
/
$ cd             (volta para casa)
$ pwd
/i1/paginas/data/wd/dracula
```

Para listar os arquivos que estão em um diretório use **`ls`** (*list*). Diferentemente do `dir` do DOS, A listagem é simples e relaciona apenas os nomes dos arquivos. Uma listagem mais longa e detalhada pode ser obtida usando a opção **`-l`** (*long listing*):

```
$ ls -l
total 4
drwxr-xr-x   2 dracula  paginas      1024 Mar  9 03:14 admin
```

```
drwxr-xr-x   3 dracula  paginas      1024 Feb 28 01:50 vampiros1
drwxr-xr-x   2 dracula  paginas      1024 Mar 14 07:41 vampiros2
-rw-r--r--   1 dracula  paginas       989 Feb 16 07:13 index.html
$
```

A maior parte dos comandos *Linux* aceitam opções que são precedidas por um traço (as do DOS eram precedidas por uma barra). A quantidade de opções por comando é muito grande, mas poucas são necessárias para a maior parte dos usuários. A opção `-l` não mostra todos os arquivos. Alguns estão ocultos. Usando a opção **`-a`** (*all*) obtém-se a lista completa. As duas opções podem ser agrupadas, por exemplo:

```
$ ls -la
total 7
drwxr-xr-x   5 dracula  paginas      1024 Mar  9 03:11 .
drwxrwxrwx  20 root     root         1024 Mar 14 12:02 ..
-rw-r--r--   1 dracula  paginas        21 Mar  9 03:30 .htpasswd
drwxr-xr-x   2 dracula  paginas      1024 Mar  9 03:14 admin
drwxr-xr-x   3 dracula  paginas      1024 Feb 28 01:50 vampiros1
drwxr-xr-x   2 dracula  paginas      1024 Mar 14 07:41 vampiros2
-rw-r--r--   1 dracula  paginas       989 Feb 16 07:13 index.html
$
```

lista os arquivos do diretório atual, de forma longa mostrando os arquivos ocultos. O `ls -la` é a forma equivalente ao `dir` do MS-DOS. Além das informações sobre tamanho, data e hora da criação do arquivo e nome (últimas quatro colunas), a listagem do *Unix* ainda contém dez caracteres que representam a função do arquivo e suas permissões de acesso e execução (primeira coluna), a quantidade de arquivos que contém (se for diretório) ou ao qual está vinculado (segunda coluna), o dono do arquivo (`dracula`) e o grupo de usuários ao qual pertence (paginas).

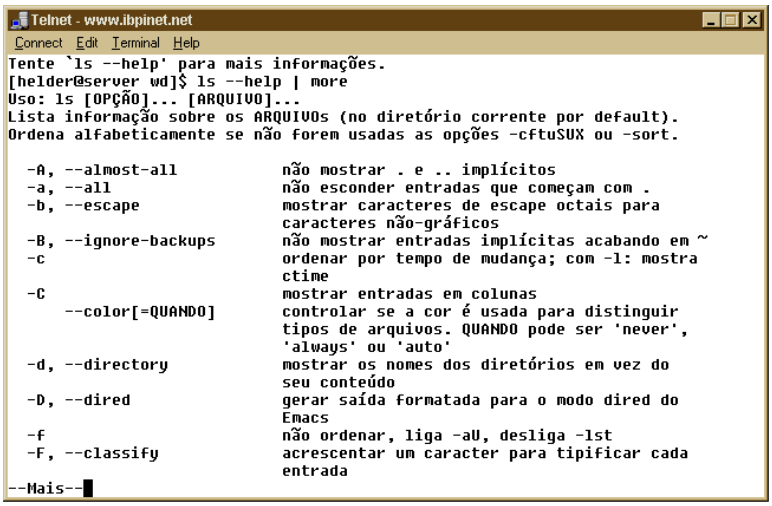
Há muito mais opções em `ls`. Para saber quais são as outras digite:

```
$ ls --help
```

Assim como em MS-DOS, `ls` também aceita argumentos (nomes de arquivos, diretórios) e os nomes de arquivo podem ser identificados com filtros `*` e `?`. A instrução:

```
$ ls -l *.ba?
```

lista apenas os arquivos do diretório que terminam em `.bak`, `.bat`, `.baz`, etc.



Tela mostrando opções do `ls` (use `ls -help | more`)

Um disco em *Linux* não tem a importância que tem em *Windows*. É representado por um arquivo geralmente localizado abaixo do diretório de sistema `/dev/`. Não existe um *drive* padrão como em *Windows*. Todos estão abaixo de `/`. Diretórios que fazem parte do sistema de arquivos de mais de um disco físico podem estar distribuídos pelo sistema de arquivos principal, e localizados fora do `/dev`.

Permissões

A listagem de arquivos mostrada por `ls -l` apresenta várias informações importantes, entre as quais estão o *tipo* do arquivo (*papel* exercido por ele no sistema operacional) e as suas *permissões*. A linha que informa a função e permissões de um arquivo contém 10 letras ou traços. A primeira letra representa a *função* do arquivo. As outras nove indicam o acesso de *leitura*, *alteração* e *execução* para usuário, grupo e os outros, respectivamente. Por exemplo, um arquivo identificado com a linha:

```
d rwx r-x r--
```

é um diretório ("d"), pode ser lido (`r`), alterado (`w`) e executado (`x`) pelo usuário dono (`rwx`), pode ser lido e executado mas não alterado pelos usuários do seu grupo (`r-x`) e não pode sequer ser executado (ter seu conteúdo listado) pelo restante dos usuários (`r--`). Se você é o dono de um arquivo, pode alterar suas permissões e restringir ou ampliar o seu acesso usando o comando **`chmod`**. Uma das formas de tornar um arquivo executável para todos é fazer:

```
$ chmod a+x nomearquivo
```

O "`a`" é de *all* (todos). O "+" indica que se está ligando o bit executável ("`x`") desse arquivo. Pode-se ainda usar "`g`" de grupo "`u`" de usuário (dono) e "`o`" de outros e preceder o bit que se quer alterar (`r`, `w` ou `x`) por `-`, caso se queira remover uma permissão. Veja mais alguns exemplos:

```
$ chmod u+w arquivo.txt   (o dono pode alterar o arquivo)
$ chmod g-r arquivo.txt   (o grupo não pode mais ler esse arquivo)
$ chmod a+rw arquivo.txt  (todos podem ler e alterar o arquivo)
$ chmod o-x programa.pl   (os outros não podem executar o arquivo)
```

Uma outra forma de usar `chmod` é através de um número de três dígitos. Cada número corresponde respectivamente às permissões de dono, grupo e outros e é resultante da soma dos valores 1 para o bit "`x`", 2 para o bit "`w`" e 4 para o bit "`r`". Por exemplo:

```
$ chmod 755 programa.pl
```

muda o padrão de bits do arquivo para `rwx r-x r-x` (4+2+1, 4+0+1, 4+0+1). Veja outros exemplos:

```
$ chmod 644 texto.txt (padrão rw- r-- r--) (4+2+0, 4+0+0, 4+0+0)
$ chmod 400 texto.txt (padrão r-- --- ---) (4+0+0, 0+0+0, 0+0+0)
$ chmod 734 progr.pl  (padrão rwx -wx r--) (4+2+1, 0+2+1, 4+0+0)
$ chmod 777 progr.pl  (padrão rwx rwx rwx) (acesso total para todos)
```

No Windows, arquivos executáveis são identificados por sua extensão (`.exe`, `.com`, `.bat`). No *Unix*, eles são identificados unicamente pelo bit `x`. Programas usados pelo servidor Web (programas CGI) têm que ser marcados como executáveis para que funcionem.

Criação e destruição de arquivos

A criação e remoção de diretórios em *Linux* usa os mesmos comandos que em *Windows*: **`mkdir`** e **`rmdir`**, respectivamente. O comando **`mkdir`** (*make directory*) cria um novo diretório abaixo do diretório atual.

```
$ mkdir temporario
$ cd temporario
$ mkdir subdir1
$ mkdir subdir2
```

Os comandos acima criam a seguinte estrutura de diretórios:

```
temporario/
        |___ subdir1/
        |___ subdir2/
```

Um diretório *vazio* sempre pode ser removido usando **rmdir**. Se ele contiver arquivos ou subdiretórios não será possível removê-lo. É preciso apagar todos os arquivos e remover todos os seus diretórios antes que se possa removê-lo. Arquivos comuns podem ser apagados usando **rm** (*remove*):

```
$ cd temporario/subdir1   (entra em temporario/subdir1)
$ rm *                    (apaga todos os arquivos)
$ cd ..                   (sobre na árvore de diretórios p/ temporario/)
$ rmdir subdir1           (remove temporario/subdir1)
```

A remoção de arquivos em *Linux* não faz perguntas (mas o administrador do seu sistema pode tê-lo configurado para que faça). Para que ele pergunte se você tem certeza, é preciso usar a opção -i:

```
$ rm -i lixo.txt
```

O *Linux* também possui um devastador removedor recursivo de arquivos e árvores de diretórios. O comando rm com a opção **-r** remove recursivamente (entra em cada diretório e repete o comando) um diretório e todo o seu conteúdo e sequer faz qualquer pergunta:

```
$ cd                  (volta para diretório casa)
$ rm -r temporario    (apaga toda a árvore de diretórios e seu conteúdo)
```

Os comandos para copiar, mover e mudar o nome de um arquivo são **mv** e **cp**. Arquivos podem ser movidos de lugar usando mv:

```
$ mv ./local.txt ~/arquivos/remoto.txt
```

O til (~) é um atalho para o seu diretório casa. Tanto faz fazer

```
$ cd
```

como

```
$ cd ~
```

Ambas as instruções voltam para o mesmo lugar.

O comando mv também serve para mudar o nome de um arquivo:

```
$ mv importante.txt inutil.txt
```

O comando **cp** é usado para copiar um arquivo de um lugar para outro (mantendo o original intacto). O comando cp requer dois argumentos: o primeiro é a origem e o segundo o destino. A instrução:

```
$ cp ./*.txt /textos/
```

copia todos os arquivos que têm extensão .txt do diretório atual para o diretório textos, situado na raiz do sistema *Unix* (se você tiver permissão de escrita em /).

O uso do diretório "." nem sempre é opcional em *Linux*. Se você tem um arquivo executável no seu diretório, ele poderá não ser executado quando você digitar o nome dele da forma:

```
$ meuprog
```

O sistema poderá não encontrá-lo pois só procura por comandos em outro lugar do disco. Para deixar claro ao sistema que você pretende executar o programa que está no diretório atual (e não outro qualquer), é preciso informar o diretório do programa (o atual) usando "**.**":

```
$ ./meuprog
```

Se `meuprog` realmente for um programa executável, e você tiver permissão para executá-lo (o bit `x` está ligado), ele irá executar.

Listagem do conteúdo de arquivos e redirecionamento

Para listar o conteúdo de arquivos existe o comando **cat**, que também serve para concatenar os arquivos (antes de listá-los). Se o arquivo for muito longo, `cat` pode ser executado através de **more** (como no MS-DOS) para que apenas uma página seja exibida de cada vez:

```
$ cat texto_longo.txt | more
```

O redirecionamento também pode ser utilizado para criar arquivos, ler o conteúdo de arquivos e acrescentar informação em arquivos existentes usando os símbolos **>**, **<** e **>>**:

```
$ cat texto.txt > copia.txt
$ filtro < dados.txt > resultados.txt
$ adiciona >> controle.txt
```

O *Linux* possui alguns arquivos especiais que são ligados a dispositivos como a impressora, o disco, o teclado e a tela (*entrada* e *saída* padrão). Os dispositivos são arquivos e seu nome depende do sistema *Unix* utilizado. Na maior parte das distribuições *Linux*, os dispositivos ficam abaixo do diretório de sistema `/dev`. Para copiar um arquivo para um disquete (formatado em *Linux*) você poderia fazer:

```
$ cp texto.txt /dev/fd0
```

Desde que o sistema de arquivos do *drive* da máquina esteja *montado* e *vinculado* ao arquivo de sistema `/dev/fd0`.

Para redirecionar da entrada padrão para um arquivo, pode-se fazer:

```
$ cat > texto.txt
```

O comando acima copiará tudo o que o usuário digitar na tela para o arquivo texto.txt até que o ele digite a combinação de teclas *Ctrl-C* (ou *Ctrl-Z* ou *Ctrl-D*, dependendo das configurações do sistema e do *Telnet*).

Para editar arquivos você pode usar os editores do *Unix* ou criá-los em *Windows* para depois transferi-los via FTP para a sua conta *Unix*. Se você desejar usar o *Linux* para editar textos, pode usar o editor **joe** (semelhante ao processador *WordStar* for DOS) ou o **pico** que são razoavelmente simples e presentes na maior parte das distribuições do *Linux*. O *Telnet* do Windows pode apresentar problemas ao exibir a tela criada por esses editores. Se você utilizá-los com freqüência, considere instalar um cliente *Telnet* mais eficiente, como o *NetTerm*.

Esses dois editores não estão disponíveis em todas as versões do *Unix*. Os editores nativos do *Unix*, como o **vi** e o **emacs** são bastante complexos e requerem um certo esforço para aprender a usá-los.

Mudança de senha e encerramento da sessão

Você pode mudar a sua senha a qualquer momento a partir da linha de comando usando o comando **passwd**. Ele pedirá sua senha antiga e para você repetir duas vezes a nova senha. Na próxima vez em que você entrar no sistema, precisará usar a nova senha. Este sistema poderá não estar disponível em alguns sistemas devido a restrições de segurança.

```
$ passwd
Changing password for dracula
(current) UNIX password: (não aparece)
New UNIX password: (não aparece)
Retype new UNIX password: (não aparece)
passwd: all authentication tokens updated successfully
$
```

ADVERTÊNCIA: No *IBPINET* este mecanismo se comporta como se funcionasse mas na verdade ele *não funciona*. Portanto, *não o utilize* para mudar a sua senha do *IBPINET*. Para alterar sua senha no *IBPINET*, use a interface Web disponível na área VIP dos assinantes.

Para *encerrar* sua sessão você pode:

- fechar a conexão *Telnet*;
- digitar o comando **logout**; ou
- apertar a seqüência *Ctrl-D*.

Variáveis de ambiente

Assim como o MS-DOS, os sistemas *Unix* também utilizam variáveis de ambiente para personalizar e definir configurações iniciais ao ambiente de linha de comando.

Uma variável definida em linha de comando irá durar enquanto a sessão (janela ou conexão *Telnet*) estiver ativa e poderá ser chamada quantas vezes for necessário para recuperar o valor guardado. Por exemplo, para armazenar um comando que será repetido várias vezes, pode-se fazer:

```
$ COMANDO="ls -l"
```

Para usar a variável de ambiente, ela deve ser chamada com o símbolo **$**:

```
$ $COMANDO
total 4
drwxr-xr-x   2 dracula  paginas      1024 Mar  9 03:14 admin
drwxr-xr-x   3 dracula  paginas      1024 Feb 28 01:50 vampiros1
drwxr-xr-x   2 dracula  paginas      1024 Mar 14 07:41 vampiros2
-rw-r--r--   1 dracula  paginas       989 Feb 16 07:13 index.html
$
```

A instrução **echo** imprime o conteúdo da variável ou o texto que recebe como argumento e pode ser usada para se ler o conteúdo de variáveis de ambiente:

```
$ echo O comando guardado é \"$COMANDO\"
```

O comando acima imprimirá:

```
O comando guardado é "ls -l"
```

Você pode listar todas as variáveis de ambiente definidas para a sua sessão atual usando a instrução `set` sem argumentos:

```
$ set
BASH=/bin/bash
COLUMNS=80
COMANDO=ls -l
```

```
HOME=/i1/paginas/data/helder
HOSTNAME=server.ibpinetsp.com.br
LC_CTYPE=ISO-8859-1
PATH=/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
PWD=/i1/paginas/data/helder
SHELL=/bin/bash
(... aqui dezenas de outras variáveis ...)
USER=helder
```

Observe a variável que nós definimos em negrito. A variável `PATH` (também em negrito) é geralmente definida no arquivo `/etc/profile` do sistema (que age como um tipo de `AUTOEXEC.BAT`) e informa onde ele deve procurar pelos programas que são digitados na linha de comando. (Ele pode ser sobreposto pelo arquivo `.profile` que o usuário pode criar e armazenar em seu diretório casa.) A maior parte dos comandos do *Linux* são aplicações localizadas nos diretórios `/bin` e `/usr/bin/`. É possível usar outros comandos localizados em diretórios que não estão no `PATH` desde que se informe o seu caminho. Como o diretório atual não faz parte do `PATH` acima, não é possível executar programas armazenados no diretório atual a não se que se informe o caminho:

```
$ ./programa (. é o caminho do diretório atual)
```

É possível redefinir o `PATH`, mas se você não fizer isto no `.profile`, a mudança valerá apenas para a sessão atual do *Shell*.

O *Unix* distingue maiúsculas de minúsculas. Não é a mesma coisa usar `SET` ou `set`, listar o arquivo `a.txt` não é a mesma coisa que listar o arquivo `A.TXT`.

Aplicações de linha de comando

As aplicações que rodam na linha de comando são como os comandos nativos do *Linux*. Para executá-las, basta digitar o nome completo do programa (inclusive extensões de nome de arquivo), quaisquer parâmetros necessários e digitar *Enter*. O diretório deve necessariamente estar no `PATH`. Se não estiver, será preciso informar o caminho do arquivo mesmo que ele esteja no diretório atual. O arquivo também deve conter código executável e ter permissão de execução.

Roteiros Shell

Se você tiver uma seqüência de instruções para realizar em linha de comando *Linux*, pode criar um *Shell script* (roteiro *Shell*). Ele é semelhante ao arquivo de lote do MS-DOS. Esse programa é um simples arquivo de textos contendo comandos *Shell* que devem ser executados em seqüência. Para criá-lo, abra um editor de textos (como o `joe` do *Linux*), digite os comandos, declarações de variáveis e/ou parâmetros desejados, salve o arquivo e execute-o digitando o nome do interpretador *Shell* (`/bin/sh`) e o seu nome na linha de comando. Por exemplo, um arquivo chamado `rotinas.txt` contendo comandos *Shell* poderia ser executado da seguinte maneira:

```
$ sh rotinas.txt
```

Não é preciso usar `/bin/sh` porque o diretório `/bin` faz parte do `PATH`.

Suponha que você precise criar um conjunto de diretórios com nomes diferentes mas contendo os mesmos subdiretórios e arquivos. Você poderia repetir a seqüência de comandos:

```
mkdir jamaica
cd jamaica
mkdir imagens
```

```
mkdir paginas
echo > index.html
```

A instrução `echo` sem argumentos imprime uma linha em branco. Redirecionar uma linha em branco para um arquivo é criar um arquivo vazio.

Criando um roteiro `criadir` com os comandos acima, podemos executá-lo através do interpretador `/bin/sh`, como mostrado acima, ou transformá-lo em um arquivo executável, para que possa ser executado da forma:

```
$ ./criadir haiti
$ ./criadir guatemala
```

e ter cada programa repetir todos os comandos do roteiro. Para transformar um roteiro Shell em um programa executável é preciso informar, na primeira linha, o caminho até o programa que deverá interpretar as linhas seguintes. Isto é feito escrevendo na primeira linha do programa, o texto:

```
#!/caminho/para_o/interpretador
```

Para receber parâmetros de linha de comando (o nome `haiti`, `guatemala`, etc) um roteiro pode usar as variáveis `$1`, `$2`, etc. `$1` corresponde ao primeiro argumento após o nome do arquivo, `$2` ao segundo e assim por diante. O conteúdo do arquivo, portanto, pode ser:

```
#!/bin/sh
mkdir $1
cd $1
mkdir imagens
mkdir paginas
echo > index.html
cd ..
```

A primeira linha do programa acima informa o nome e caminho até o interpretador *Shell*. Esse interpretador irá executar cada uma das linhas seguintes. O *Unix* dispõe de outras linguagens de *Shell* que podem ser usadas como o *C Shell* e o *Korn Shell*. O *Shell* que estamos usando é chamado de *Bourne Shell*. Ao se executar:

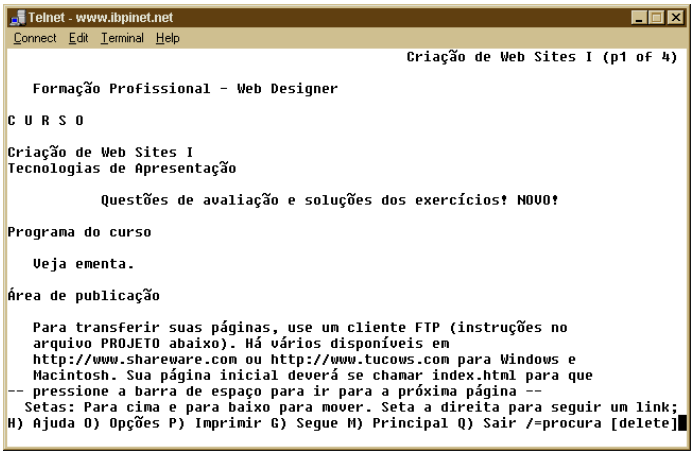
```
$ ./criadir japao
```

A variável `$1` será substituída pelo valor `japao` e será criado um diretório `japao` contendo os subdiretórios `paginas`, `imagens` e um arquivo `index.html` vazio.

O arquivo `.profile` é um roteiro que é chamado automaticamente pelo *Shell* do seu sistema para definir os parâmetros de inicialização do seu ambiente e rodar os comandos e aplicações que você solicitar na linha de comando. Se você não quiser usar o *Bourne Shell* poderá mudar para outro como o *C Shell*, simplesmente digitando:

```
$ csh
```

Os comandos básicos são os mesmos mas o *C Shell* tem comandos adicionais e recursos que permitem buscas e substituições mais sofisticadas. Para sair de um *Shell* use o comando **`exit`**.



Aplicações de rede

No *Unix* existem várias aplicações de linha de comando que

funcionam em rede, requerendo uma conexão ativa para funcionarem perfeitamente. Vários servidores estão no ar permanentemente. Você pode usar o sistema *Unix* para conectar-se via *Telnet* a outro sistema remoto. Pode também usar um cliente FTP orientado a caracter (com os mesmos comandos do cliente que roda no MS-DOS).

Um browser orientado a caractere está disponível no sistema *Linux* do IBPINET. É o *Lynx*. Com ele você pode navegar pela Internet. A sua interface não funciona corretamente através do *Telnet* do *Windows*. Use um cliente *Telnet* como o *NetTerm* para navegar como nos velhos tempos da Web (veja figura).

2.3. Exercícios

Linha de Comando do Windows (MS-DOS)

1. Configure o `AUTOEXEC.BAT` para que ele rode o programa `doskey`. Reinicie o *Windows*. Abra agora uma janela do MS-DOS e digite alguns comandos. Você pode agora recuperar os últimos comandos digitados com a seta para cima e usar a tecla *Ins* para inserir alterações em comandos já digitados.
2. Abra a janela do Prompt do MS-DOS, use `cd` para ir até a pasta onde você tem armazenados seus arquivos HTML (`wd\`)e liste o seu conteúdo. Observe a diferença entre os nomes do DOS e do Windows.
3. Crie um subdiretório chamado `temp`, dentro do diretório onde estão seus arquivos HTML. Entre nesse diretório.
4. Copie os arquivos HTML (`*.html`) do diretório anterior para o subdiretório `temp` (diretório atual).
5. Copie quaisquer arquivos `*.jpg` do diretório anterior para o subdiretório `temp` (atual).
6. Crie dois subdiretórios abaixo de `temp`, de forma a obter a seguinte árvore:

```
temp\
    |___ imagens\
    |___ html\
```

7. Mova os arquivos JPG do diretório atual (`temp`) para dentro do subdiretório imagens e os arquivos HTML do diretório atual para o subdiretório `html`. Abra as pastas correspondentes no *Windows* e veja o resultado.
8. Entre no subdiretório HTML e liste o conteúdo de uma de suas páginas HTML. Use more para que você possa ver o conteúdo página por página.
9. Crie um novo arquivo `comandos.txt`, usando `copy con:` e redirecionamento. O novo arquivo deve conter um comando para renomear todos os arquivos `*.html` em arquivos `.txt`.
10. Transforme o arquivo comandos.txt em um arquivo de lote e execute-o. Abra a pasta atual no *Windows* e veja agora como o *Windows* representa esses arquivos.
11. Imprima a variável de ambiente PATH na sua tela.
12. Crie um arquivo de lote chamado `tipos.bat` que construa três subdiretórios abaixo do subdiretório imagens. Esses subdiretórios devem todos ter o mesmo prefixo, e terminar em `_jpg`, `_gif` e `_png`. Deixe que o usuário do programa escolha o prefixo. Por exemplo, se ele escolher o prefixo `foto1`, ele deve poder executar sua aplicação da seguinte forma:

    ```
    tipos foto1
    ```

    e a aplicação deverá criar os subdiretórios `foto1_jpg`, `foto1_gif` e `foto1_png`. Teste a aplicação.
13. Apague toda a árvore de diretórios que você construiu nestes exercícios.

Unix Shell (Linha de Comando no Linux)

14. Conecte-se remotamente via telnet a um sistema *Unix* (ou o sistema *Linux* do IBPI. Você pode se conectar ao sistema *Linux* do IBPINET, se você tiver discado para o provedor do IBPINET (se você estiver em outra rede, seu acesso será negado). Você deve usar a porta 44500:

    ```
    telnet www.ibpinet.net 44500
    ```

15. Uma vez conectado, digite seu nome e senha para ter acesso ao sistema.
16. Veja qual o seu diretório atual no *Unix*. Navegue pelo sistema, suba até a raiz, e liste os arquivos. Observe os nomes de arquivo, os tipos, as permissões e usuários. No final de sua "viagem", volte para o seu diretório casa.
17. Liste os arquivos do seu diretório casa. Veja se há algum arquivo oculto.
18. Crie um subdiretório `aprendix` abaixo do seu diretório casa. Entre nele.
19. Crie a seguinte árvore de diretórios abaixo de `aprendix`:

```
aprendix\
    |___ figuras\
    |___ paginas\
```

20. Copie todas as páginas `*.html` do seu diretório casa para dentro de `aprendix`. Copie também imagens GIF e JPG.
21. Mova as páginas de `aprendix` para o subdiretório paginas. Mova as imagens para o subdiretório imagens. Liste o conteúdo de cada diretório.
22. Entre no diretório páginas e mude as permissões de alguns arquivos. Desabilite a permissão de alteração de um arquivo (para o usuário dono) e tente apagá-lo. Desabilite a permissão de leitura e tente lê-lo (usando `cat`, por exemplo).
23. Crie um novo arquivo comandos usando `cat` e redirecionamento. O novo arquivo deve conter um comando para mudar o nome de todos os arquivos `*.html` em arquivos `.txt`.
24. Transforme o arquivo comandos em um roteiro *Shell*, mude suas permissões para permitir a execução e execute-o. Liste o conteúdo do diretório para verificar se o programa funcionou.
25. Imprima a variável de ambiente `PATH` na sua tela.
26. Crie um roteiro *Shell* chamado `tipos` que construa três subdiretórios abaixo do subdiretório `imagens`. Esses subdiretórios devem todos ter o mesmo prefixo, e terminar em `_jpg`, `_gif` e `_png`. Deixe que o usuário do programa escolha o prefixo. Por exemplo, se ele escolher o prefixo `foto1`, ele deve poder executar sua aplicação da seguinte forma:

    ```
    tipos foto1
    ```

    e a aplicação deverá criar os subdiretórios `foto1_jpg`, `foto1_gif` e `foto1_png`. Teste a aplicação.
27. Apague toda a árvore de diretórios que você construiu nestes exercícios.
28. Encerre a sessão.

3. Servidor Web

3.1. O que é um servidor Web

Chamamos de servidor Web o software que, executando, torna uma máquina capaz de oferecer o serviço de transferência de arquivos via protocolo HTTP através de uma determinada porta TCP/IP. A porta geralmente é a de número 80, mas um software de servidor pode ser instalado e configurado para operar em qualquer outra. Depois de iniciada a execução, a máquina está pronta para receber requisições de clientes HTTP, que irão se conectar à porta do servidor e fazer requisições, solicitando arquivos, imagens e outros recursos. O servidor atenderá aos pedidos e imprimirá o conteúdo dos arquivos solicitados na sua porta de serviços, de onde o cliente pode recuperá-los.

A função básica do servidor Web é essencialmente gerenciar um sistema de arquivos e responder às solicitações dos clientes, geralmente pedindo arquivos localizados nesse sistema. O servidor geralmente não lê ou interpreta os arquivos que devolve.

Comunicação entre agentes Web

Clientes e servidores Web comunicam-se entre si utilizando o protocolo de comunicação HTTP. Um cliente qualquer faz um pedido ao servidor e este retorna alguma coisa referente a este pedido. No caso da Web, o *browser*, que é o cliente Web, envia um pedido ao servidor através de uma instrução contento um *método* do protocolo HTTP (por exemplo, o método GET – solicita uma página HTML). O servidor entende o pedido, verifica a possibilidade de executá-lo e envia uma resposta que pode ser o recurso solicitado ou uma página informando o motivo pelo qual não foi possível atender o pedido. Informações adicionais são passadas através de *cabeçalhos* que são utilizados por ambas as partes. Eles precedem os dados enviados pelo servidor e seguem a linha de comando (que contém o método) enviado pelo browser. Esses cabeçalhos podem definir *variáveis de ambiente* que atuam dentro do escopo da conexão (visíveis apenas pelo processo do servidor) e possibilitam operações interativas como aplicações CGI. Os cabeçalhos são linhas de texto no formato *Propriedade: valor*. Esse formato dos cabeçalhos é padronizada na especificação Internet RFC822 que regula o formato dos cabeçalhos de mensagens de e-mail (que possuem o mesmo formato).

A tarefa do servidor é localizar o arquivo, identificar o seu tipo de dados, montar um cabeçalho que contenha informações sobre ele (tipo de dados, tamanho, data de modificação, etc.) e enviar as informações para a saída padrão, onde podem ser recuperadas pelo browser. O servidor não analisa o *conteúdo* do arquivo mas simplesmente o redireciona à porta HTTP, portanto ele não precisa conhecer nem ser capaz de interpretar HTML.

Quando um visitante clica em um link ou quando um usuário de browser digita uma nova URL na barra de endereços, o browser envia uma instrução de *requisição* para o servidor. O servidor, por sua vez, sempre devolverá uma *resposta*, havendo ou não sucesso. Uma transação típica é a seguinte:

O usuário selecionou a URL `http://volans.argo.net/book/ch1.html`. O browser, então, constrói a seguinte requisição e a envia para o servidor:

```
GET /book/ch1.html HTTP/1.0
Host: volans.argo.net
Port: 80
Accept: text/html
Accept: image/jpg
User-Agent: Microsoft Internet Explorer (Mozilla 3.0 Compatible)
```

A primeira linha é a linha da requisição. O nome em negrito é o método, que precede a URL absoluta (relativa à máquina servidora). No final da linha está o protocolo que o browser pretende usar para realizar a comunicação com o servidor (HTTP, versão 1.0). As linhas seguintes contém um possível cabeçalho para esta requisição. Isto é o que ocorre nos bastidores quando você clica em um link.

Se o servidor *encontrar* o arquivo solicitado, retornará a seguinte resposta:

```
HTTP/1.0 200 OK
Date: Friday, June 13, 1977
(... outras linhas de cabeçalho...)
Content-type: text/html

<HTML><HEAD>
<TITLE> Capitulo 3</TITLE>
(...)
```

A primeira linha é a linha da resposta, que começa com o protocolo (e versão) que o servidor está usando para responder à requisição. Em negrito estão o código e a mensagem sobre o estado da conexão. *200 OK* indica sucesso. As linhas de cabeçalho contêm informações que descrevem os dados que seguem o bloco dos cabeçalhos, que termina com uma linha em branco. Se o servidor não achar o arquivo solicitado, devolverá uma resposta diferente:

```
HTTP/1.0 404 Not Found
Date: Friday, June 13, 1977
(... outros cabeçalhos...)
Content-type: text/html

<HTML><HEAD>
<TITLE>404 File Not Found</TITLE>
(...)
```

O servidor sempre retorna algo para o cliente após receber uma requisição. A resposta pode ser o recurso que ele de fato pediu ou uma mensagem de erro, devolvida na forma de uma página HTML como no exemplo acima. Se o arquivo solicitado for uma página HTML, uma imagem GIF ou qualquer outro arquivo suportado pelo browser, este saberá como exibi-lo, pois o bloco do cabeçalho informará o seu tamanho e o seu tipo MIME. No caso de outros formatos, o browser ou redireciona para outra aplicação, ou tenta salvar os dados recebidos em disco.

Requisição para execução de programas

Se estiver configurado para oferecer suporte a *programas*, o servidor poderá, ao invés de simplesmente encontrar um arquivo e remetê-lo ao browser, tentar executá-lo. Um programa controlado pelo servidor, ao ser executado, pode comunicar-se com outros programas permitindo que o browser controle aplicações remotas através do servidor.

A forma mais comum de suporte à aplicações executadas via servidor é CGI, ou *Common Gateway Interface* (Interface Comum de *Gateway*). É um padrão W3C suportado por todos os servidores Web e estabelece parâmetros que possibilitam a execução de aplicações através dele. O arquivo do programa CGI necessita ser identificado e chamado pelo browser via uma requisição HTTP da mesma forma como ocorre com um arquivo HTML comum. A diferença é que o servidor, ao encontrá-lo, tentará executá-lo em vez de simplesmente redirecioná-lo para a saída padrão. Para que o uso de CGI seja possível, é preciso que o servidor e as aplicações CGI estejam *configurados* para funcionar dessa forma.

Este curso pretende mostrar como *implantar* CGI no servidor e como modificar aplicações simples para que atendam aos requisitos necessários para funcionarem como *programas* CGI. Não serão explorados recursos específicos das *linguagens* usadas com CGI.

Software de servidores Web

Os principais programas de servidores Web rodam em sistemas *Unix* e *Windows NT*. Atualmente (2000), o servidor mais popular é o *Apache*, que roda principalmente em sistemas *Unix*. Segundo a *NetCraft* (empresa que realiza pesquisas sobre servidores), os sites na *Internet* servidos pelo servidor *Apache* correspondem a mais de 60%. Em segundo lugar estão os servidores Microsoft, principalmente o *Internet Information Server*, líderes na plataforma *Windows*.

3.2. Configuração

O servidor possui várias propriedades que podem ser alteradas após a instalação como a *porta de serviços* (que nem sempre pode ser a de número 80), o *diretório raiz de documentos* (localização do diretório correspondente à "/" no servidor, *arquivos de índice* ou *documentos padrão* (arquivos que são enviados automaticamente quando o cliente solicita uma URL terminada em "/"). Pode ainda definir quais os *tipos de arquivos* suportados e como são localizados a partir de extensões de nome de arquivo, realizar o *redirecionamento* de URLs, definir *controle de acesso*, *mapeamento* de URLs e diretórios e instalação de *aplicações lado-servidor* (como programas CGI utilizados para responder a formulários) . Todos esses parâmetros podem ser alterados configurando-se o servidor.

A configuração de um servidor pode ser através de uma aplicação gráfica, com menus, janelas, botões, etc., através de um browser ou através da alteração de arquivos. O primeiro tipo é mais comum entre os servidores mais populares da plataforma *Windows*. O *Apache*, servidor Web mais usado no mundo, roda em *Unix* e pode ser configurado alterando parâmetros declarados em arquivos de texto.

Na seção seguinte será mostrado como configurar dois servidores. Antes, precisamos conhecer algumas propriedades configuráveis.

Porta de serviços

A porta de serviços determina a porta TCP/IP que processo do servidor Web ficará *escutando* a espera de requisições de um cliente. Alguns servidores pedem o número da porta desejada durante a instalação. Outros, escolhem sem perguntar, a porta 80, mas geralmente o administrador pode trocá-la por outra porta depois. se for necessário.

Arquivos de índice

Uma URL contém o protocolo, a máquina e o caminho até um determinado arquivo na Internet. Freqüentemente digitamos apenas o protocolo e a máquina ao localizar um site, e o servidor ainda assim devolve uma página HTML. A página devolvida é chamada de arquivo índice ou página padrão. Sempre que uma URL terminar em diretório (em "/"), o arquivo índice localizado naquele diretório será devolvido. Nos servidores *Unix* o

arquivo índice geralmente se chama `index.html` ou `index.htm`. Nos servidores *Microsoft*, o nome comum é `default.htm` ou `default.asp`. Esses nomes geralmente podem ser alterados.

Raiz de documentos

A raiz de documentos de um servidor é o diretório base onde começa o seu sistema de arquivos. Essa raiz não é a mesma coisa que a raiz do sistema. Geralmente corresponde a um certo diretório dentro do sistema que foi escolhido para guardar as páginas que serão servidas pelo servidor. Por exemplo, se a raiz de documentos de um servidor é

```
C:\Apache\htdocs\
```

Tudo o que estiver fora de `htdocs` será inacessível para quem estiver chegando à máquina via servidor Web, pois esse diretório será considerado "`/`" para o servidor.

Aliases ou mapeamentos

Aliases são usados para incluir diretórios que estão fora da raiz de documentos e para reorganizar a árvore de arquivos e diretórios do servidor.

O sistema de arquivos do servidor Web não é necessariamente um espelho de um subsistema de arquivos da máquina onde reside o servidor. Pode ter uma hierarquia bem diferente da estrutura de arquivos no disco. Pode-se definir por exemplo diretórios de mesmo nível no disco como tendo uma estrutura hierárquica no servidor:

```
c:\raiz      /
c:\docs      /livro
c:\extras    /livro/anexos
```

No exemplo acima, a primeira coluna representa a estrutura física de três arquivos na máquina onde residem. Têm o mesmo nível. Na coluna seguinte estão as URLs, onde os diretórios foram organizados de forma hierárquica. Se você estiver conectado via Web lendo uma página que está em `/livro/anexos/`, ela *fisicamente* (do ponto de vista do sistema operacional) estará disponível em `c:\extras`. Se a página requisitar uma imagem em `../abc.gif`, a imagem será solicitada pelo browser em `/livro/abc.gif`, mas estará fisicamente em `c:\docs`. O caminho "`..`" para o servidor, foi, portanto, diferente do caminho "`..`" do sistema operacional.

Mapeamentos também são usados para mapear URIs e outros domínios.

Diretórios executáveis

Um servidor pode mapear diretórios e rotulá-los como executáveis. Assim, quando um cliente solicitar um recurso armazenado naquele diretório, o servidor não o devolverá para o browser mas o tentará executar.

Diretórios marcados como executáveis são necessários para permitir a execução de programas CGI e outros tipos de aplicações Web em certos servidores.

Tipos de arquivos

É extremamente importante que o browser saiba que tipo de arquivo está recebendo. O servidor informa o tipo através de sua resposta no cabeçalho. O servidor descobre o tipo de dados de um arquivo através a extensão. Nem sempre uma extensão realmente informa o que contém. É preciso recorrer a tabelas que relacionam extensões de nome de arquivo com tipos MIME. Um arquivo com extensão `.asp`, por exemplo, é (geralmente) enviado ao browser tendo o tipo `text/html`, após o processamento por um servidor Microsoft.

Controle de acesso

É possível limitar ou negar o acesso de partes de um site a um grupo de usuários, a usuários determinados, a clientes locais (no mesmo domínio ou máquina) ou clientes localizados em determinadas redes. Dependendo das propriedades declaradas no arquivo, a resposta do servidor ao browser poderá fazer com que:

- a página seja exibida normalmente (comportamento normal);
- a página não seja exibida a menos que o cliente esteja conectado a partir de um determinado domínio;
- apareça uma janela no browser do cliente pedindo o nome e senha para acesso àquela parte do site, sempre ou quando ele estiver fora de certos domínios;
- o cliente seja redirecionado a outro servidor ou outra página;
- o conteúdo do diretório seja ou não seja exibido se não houver arquivo índice;
- certos arquivos sejam ocultados.

Por que configurar um servidor Web?

Embora o Web designer não precise entender os detalhes do funcionamento de um servidor, pois em geral não tem permissão de acesso à sua configuração, poderá se beneficiar desses conhecimentos para ter maior controle na administração do seu site sabendo localizar diretórios de programas CGI em um servidor, diretórios fora da raiz do servidor que foram mapeados em URLs abaixo da raiz de documentos e poder configurar o acesso local a partes de seu site.

Adicionalmente, o Web designer pode desejar ter um servidor Web local, instalado na sua máquina de trabalho (*Windows* ou *Mac*, possivelmente) para testar sites que usam aplicações cliente-servidor. Nesses casos, é importante que saiba alterar essa configuração. Nas duas seções a seguir veremos como instalar e configurar dois servidores Web: o *Apache*, disponível para *Windows*, é uma réplica da versão *Unix* que serve a maior parte dos sites na Web. O *Personal Web Server* é uma versão desktop limitada do *Internet Information Server*, servidor mais popular na plataforma *Windows*.

3.3. Instalação e configuração básica do Apache

O servidor *Apache* nas versões *Windows* e *Unix* está disponível em `www.apache.org`. O software é de graça e pode ser usado para qualquer fim, comercial ou não.

Para instalar o *Apache* em um sistema *Windows* é preciso ter uma infraestrutura de rede disponível. Se você nunca usa o seu computador em rede a não ser para ter acesso à Internet, é preciso instalar um adaptador de rede TCP/IP com endereço IP fixo na sua máquina. Se sua máquina possui uma placa de rede, basta adicionar o protocolo TCP/IP e configurá-lo com endereço IP, nome, etc. Se não tiver, ainda há uma solução.

Se você usa *Windows 98*, é possível configurar o ambiente necessário para o Apache instalando primeiro um outro servidor da própria Microsoft: o *PWS* (seção seguinte). Você pode depois, desinstalá-lo. Parece uma solução estranha, mas a instalação do PWS automaticamente configurará o seu sistema e definirá um nome de domínio para sua máquina, usando o nome que você deu ao seu computador (ícone "Meu Computador" na área de trabalho). Depois de instalar o *Apache* você deve desligar o PWS ou desinstalá-lo para que ele não entre em conflito com o Apache por causa da porta 80.

Você pode estar se perguntando. Por que instalar o Apache se o PWS oferece o mesmo serviço Web? A vantagem em usar o *Apache* e não o *PWS* para os exemplos e exercícios deste curso é o fato de que ele oferece mais recursos configuráveis que o *PWS*. Ele também é praticamente idêntico à sua versão *Unix*, que será usada pela maioria dos Web designers. Se você desenvolve um site hospedado em um servidor *Unix*, é útil ter um site espelho na sua máquina que use o mesmo servidor, principalmente se você desenvolve ou configura aplicações Web.

Instalação

A instalação do *Apache* no *Windows* é similar à qualquer outra instalação de programa *Windows*. Durante a instalação, o programa perguntará onde o servidor deve ser instalado. Geralmente ele tenta instalar o programa abaixo de `c:\Arquivos de Programas\...` Como vamos ter que navegar no sistema de arquivos de diretório usando MS-DOS, será muito chato ter que digitar diretórios longos entre aspas (porque MS-DOS não suporta espaços) todas as vezes que precisarmos rodar algum programa nos diretórios do servidor. Para evitar isto, mude o diretório de instalação para:

```
c:\Apache\
```

Uma vez instalado, o *Apache* pode ser inicializado a partir do menu *Iniciar*. Ele deve abrir uma janela do DOS. Se isto não ocorrer ou se a janela abrir e depois fechar, provavelmente ocorreu um erro. Para ver qual foi a mensagem de erro, abra uma janela do MS-DOS, mude para o diretório `c:\Apache\` e rode o arquivo `apache.exe`. Você provavelmente verá a mensagem de erro. Os erros mais comuns são:

- A falta de infraestrutura de rede (sua máquina não tem TCP/IP). Se sua máquina não tem ambiente de rede disponível no Windows ou se tem, mas não instalou o protocolo TCP/IP no adaptador de rede, o Apache poderá não rodar. Tente a solução via instalação prévia do PWS, descrita acima, pois ele copiará os arquivos necessários à infraestrutura que o Apache precisa.
- A falta de um nome TCP/IP para a máquina (nome do *host* local). O Apache tentará descobrir o nome de sua máquina, se ela estiver usando um nome TCP/IP (DNS). O Apache poderá não descobrir o nome de sua máquina se esse nome não estiver definido na configuração TCP/IP do Ambiente de Rede, mesmo que sua máquina tenha um nome atribuído pelo Windows. Você pode usar o nome Windows do seu computador desde que altere, no arquivo httpd.conf (veja *Configuração*, adiante) do Apache, a propriedade ServerName, informando o nome de sua máquina.
- Conflito de porta (se você instalou antes o PWS e não o desinstalou ou não o desligou, haverá um conflito porque os dois servidores estarão disputando a mesma porta de serviços. Pare o *PWS* e tente outra vez. Mais adiante veremos como mudar a porta do *Apache* para que você possa rodar os dois servidores ao mesmo tempo, se quiser.

Para testar o funcionamento do servidor (se tudo estiver OK e o servidor tiver iniciado uma janela do DOS que não fechou), abra um browser e digite a URL:

```
http://localhost/
```

e você deve ver aparecer na janela do seu browser a página inicial do *Apache*.

O nome `localhost` é um nome genérico utilizado para identificar a máquina atual. Se seu computador faz parte de uma rede, você pode usar o seu nome de rede também ou seu nome de domínio TCP/IP, se instalado. Caso você não tenha rede e tenha instalado o Apache usando o "macete" da prévia instalação do *PWS*, o nome de sua máquina será o nome do seu computador (geralmente o que aparece no ícone "Meu Computador", sem espaços).

Você pode escolher um outro nome para o seu computador, quando ele for acessado através do servidor Apache. Veja na seção seguinte como fazer isto.

A maior parte da configuração básica do *Apache*, descrita a seguir, pode ser realizada tanto em servidores *Windows* como em servidores *Unix*.

Configuração básica

A configuração do *Apache* é realizada através de arquivos de configuração localizados no subdiretório `conf/` do diretório de sua instalação (`C:\Apache\conf`, `/usr/apache/conf/` são diretórios típicos).

Em algumas versões, a configuração está distribuída em três arquivos: `httpd.conf`, `srm.conf` e `access.conf`. Em outras (no *Windows*, por exemplo), tudo pode ser alterado a partir do `httpd.conf`.

O `httpd.conf` e `srm.conf` são arquivos de texto que contém várias declarações e comentários. As declarações são propriedades de podem ser alteradas, como porta de serviços, raiz de documentos e outras características do servidor. Os comentários são linhas de texto precedidas por #. Qualquer linha que começar com # é ignorada. Várias opções do servidor podem ser ligadas ou desligadas simplesmente tirando ou acrescentando o "#". Qualquer mudança nos arquivos de configuração só entra em vigor quando o servidor é reinicializado.

Abra o arquivo `httpd.conf` e dê uma olhada no seu conteúdo. Há muita coisa que pode ser configurada. A seguir, veremos apenas algumas propriedades. Para localizar uma propriedade configurável, procure-a usando a ferramenta "Localizar" do seu editor de textos. Tenha cuidado com maiúsculas e minúsculas, pois o *Apache* detecta a diferença.

Configuração: porta de serviços

A porta de serviços pode ser alterada na instrução **`Port`**. Procure pela palavra "Port", considerando as maiúsculas, que você rapidamente encontrará a linha:

```
Port 80
```

Esse é o valor *default* da porta. Se for necessário (caso tenha mais de um servidor), você poderá escolher outro número de porta:

```
Port 8080
```

A mudança de qualquer propriedade nos arquivos de configuração só entrarão em vigor na próxima vez que o Apache for iniciado.

Configuração: raiz de documentos

A raiz de documentos pode ser redefinida através da propriedade **`DocumentRoot`**. Na instalação *default* do *Apache Windows* este valor é:

```
DocumentRoot "c:/Arquivos de Programas/Apache Group/Apache/htdocs"
```

As aspas só devem ser usadas no *Apache Windows*. No *Linux* do laboratório, o caminho é:

```
DocumentRoot /usr/apache/htdocs/
```

Escolha o seu `DocumentRoot` para que aponte para o diretório onde começa o seu site. Qualquer diretório construído abaixo dele automaticamente será visível via servidor.

Configuração: documentos padrão

Você pode mudar o nome dos arquivos índice ou acrescentar outros à lista através da propriedade **`DirectoryIndex`**. Esses arquivos serão devolvidos para o browser quando a URL terminar em /:

```
DirectoryIndex index.html index.htm default.htm index.shtml
```

Na ausência de um documento padrão, o servidor poderá gerar uma página mostrando o conteúdo do diretório (que poderá omitir alguns arquivos.) Poderá também não mostrar o conteúdo do diretório, mas devolver uma página informando que tal listagem foi proibida pelo administrador do servidor. Em uma seção mais adiante veremos como mudar esse comportamento.

Configuração: mapeamentos

Você pode criar diretórios virtuais a partir do sistema de arquivos do servidor usando a propriedade **`Alias`**. Qualquer novo diretório definido estará disponível através do serviço Web e poderá apontar para qualquer outro diretório do sistema operacional, esteja ou não abaixo da raiz de documentos:

```
Alias /videos/ "c:/cinema/abril/videos/"
Alias /contas/ "c:/planilhas/web/"
Alias /imagens/gif/ "c:/figuras/"
```

Com `Alias` é possível criar uma árvore de diretórios no servidor Web que nada tenha a ver com o sistema de arquivos do sistema operacional.

Configuração: nome do servidor

Se você alterar o nome do seu computador (ou se o Apache não conseguir descobrir qual é o nome dele), você precisa definir a propriedade **`ServerName`**, que deve informar um nome válido para a sua máquina:

```
ServerName ganimede
```

Você não pode inventar qualquer nome. O nome deverá ser conhecido pelo serviço de nomes da sua rede. No Windows, você pode acrescentar um nome para o seu computador criando um arquivo `hosts` (ou utilizando um já existente) no diretório `c:\Windows\`. O arquivo deve chamar-se apenas `hosts` (sem extensões de nome de arquivo) e conter a seguinte linha, informando os nomes pelos quais a máquina pode ser identificada:

```
127.0.0.1    localhost  ganimede
```

`127.0.0.1` é um número IP reservado que refere-se à máquina local. Se sua máquina tiver um endereço IP (se ela estiver em rede com o ambiente TCP/IP configurado), você deverá usar em vez de `127.0.0.1` o endereço de sua máquina. `127.0.0.1` nesse caso deve ser usado *apenas* para o nome `localhost`:

```
127.0.0.1       localhost
200.253.191.74  ganimede  ganimede.ibpinet.net  www3.ibpinet.net
```

Qualquer um dos nomes colocados ao lado do endereço IP do arquivo `hosts` pode ser usado como seu `ServerName`.

Estatísticas de erro e acesso

O Apache guarda estatísticas de acesso e mensagens de erro em arquivos `*.log` disponíveis no subdiretório `logs/` (dentro da instalação do *Apache*). O arquivo `error.log` contém todos os erros e a hora e dia em que ocorreram. É útil para depurar o servidor e programas instalados. O arquivo `access.log` guarda todos os acessos (de páginas, imagens, etc.) incluindo informações sobre browser, endereço IP, domínio além de data e hora. É útil para gerar estatísticas e para obter informações confiáveis sobre número de acessos.

Como publicar páginas

Para publicar páginas no *Apache* basta copiar os arquivos para um diretório disponível a partir da raiz de documentos. O diretório pode estar fisicamente abaixo do diretório raiz ou pode ter sido mapeado com uma declaração `Alias`. Todos os arquivos de um diretório serão visíveis e acessíveis para download pelo browser a não ser que sejam definidas restrições de acesso.

Programas localizados em diretórios de documentos não serão executados. Serão transferidos pelo browser que poderá executá-los no cliente (se for possível) ou salvá-los em disco (*download*).

Instalação de programas

Para que um servidor possa rodar programas por solicitação do browser, eles precisam ser instalados no servidor. A instalação de programas CGI em servidores *Apache* pode ser feita de duas formas:

- através da criação de um diretório virtual especial com permissão de execução
- através da definição de uma extensão que identifique arquivos executáveis pelo servidor.

Não é interessante, do ponto de vista da segurança, permitir que qualquer programa armazenado na máquina servidora possa ser executado pelo servidor Web. A criação de um diretório executável limita a execução à programas residentes em uma certa parte da máquina. A configuração do *Apache* para uso com CGI será detalhada em outra seção.

A instalação de outros tipos de programas: ASP, PhP, servlets Java, etc. é dependente da tecnologia escolhida. Entre tais programas, alguns podem residir em diretórios de documentos.

Controle de acesso no Apache

No *Apache*, o administrador do servidor Web pode configurar o nível de acesso de todos os sites por ele servidos no arquivo de configuração `access.conf` (ou `httpd.conf`). Um usuário do sistema que não tenha acesso aos arquivos de configuração do servidor Web (o Web designer, por exemplo) pode redefinir vários dos parâmetros que limitam o acesso às áreas do site sob seu domínio usando arquivos `.htaccess`.

O arquivo `.htaccess` é um arquivo de texto comum que deve ser criado pelo usuário. Possui uma lista (geralmente pequena) de instruções que determinarão o comportamento do servidor quando algum arquivo do diretório onde reside o `.htaccess` for solicitado. Existindo um arquivo `.htaccess` em um diretório utilizado pelo servidor Web, este irá ler seu conteúdo e redefinir o seu nível de acesso, podendo sobrepor parâmetros estabelecidos pelo administrador do servidor nos arquivos `*.conf` inacessíveis à maior parte dos usuários.

São diversas as propriedades que podem ser alteradas no através de um arquivo `.htaccess`. As principais são:

- Guardar o nome e localização dos arquivos de usuários, grupos e senhas criados pelo usuário (o Web designer);
- Limitar o nível de acesso que terão os clientes que solicitarem arquivos do diretório onde reside, determinando quais usuários, grupos de usuários ou máquinas da Internet podem ter acesso aos seus recursos e conteúdo; e
- Determinar como será exibido o conteúdo de um diretório na ausência de um arquivo de índice (`index.html`, `index.htm`, ou outro).

Há outras propriedades mas que só podem ser definidas no arquivo `access.conf` (`httpd.conf`). O administrador do servidor Web, que tem acesso à configuração do servidor, também tem o poder de limitar o que pode ou não ser redefinido pelos usuários através do `.htaccess`. Existe a possibilidade de um provedor de informações configurar seu servidor Web de tal forma que ele ignore arquivos `.htaccess` definidos por usuários.

Criar um arquivo `.htaccess` é bastante simples. As propriedades que podem ser alteradas são poucas mas muito úteis. Nesta seção mostraremos as propriedades mais importantes que o Web designer pode definir usando esse arquivo especial.

Primeiro, é preciso criar um arquivo de textos (usando qualquer editor disponível). O arquivo não aparecerá quando arquivos forem listados usando `ls` ou `ls -l` em sistemas Unix. Arquivos que começam em ponto "`.`" só aparecem quando listados com `ls -la`. Os parâmetros são definidos como instruções de uma linha em uma ordem não definida. Tudo no `.htaccess` é opcional mas a declaração de certas propriedades pode exigir a declaração de outras. Eis o conteúdo de um arquivo `.htaccess` simples que apenas permite a listagem do conteúdo de um diretório (na ausência de um arquivo `index.html` ou similar):

```
Options Indexes
```

Agora o servidor irá gerar uma listagem dos arquivos do diretório na forma de uma página visível pelo cliente. Se o servidor, por *default* (na configuração), já mostra os arquivos dos diretórios que não possuem arquivos índice, pode-se impedir que ele o faça usando um `.htaccess` contendo:

```
Options None
```

A regra e restrições valem para todo o diretório e árvore de diretórios abaixo do diretório onde reside o `.htaccess`.

Um arquivo típico, usado para controlar níveis de acesso, consiste de instruções (declarações) globais (como `Options`) e uma ou mais diretivas `<Limit> ... </Limit>` que informa as restrições de acesso para certos métodos do browser. Veja um exemplo de um arquivo `.htaccess` (bem maior que o usual) para o *Apache* (no *Windows*):

```
Options None
AuthType Basic
AuthName AdminVampiros
AuthUserFile  c:\Apache\auth\usuarios.txt
AuthGroupFile c:\Apache\auth\grupos.txt

<Limit GET>
   require user morpheus
   require user malacoda
   require group vampiros
   deny from .batcaverna.com
</Limit>

<Limit POST>
   order allow, deny
   allow from .vampiros.org
   deny from all
</Limit>
```

A primeira instrução acima (`Options`) define recursos avançados disponíveis no diretório como permissão para interpretar páginas com instruções para o servidor[1] (opção `Includes` e `ExecCGI`), visualização do diretório na ausência de arquivo (opção `Indexes`) entre outras. As opções são separadas por espaços logo após a instrução `Options`. A opção `All` ativa todos os recursos especiais (desde que permitido pelo administrador do servidor). `None`, usado acima, os desliga. Veja alguns exemplos:

```
Options All  # todos os recursos ligados
Options Indexes Includes ExecCGI  # três recursos ligados
```

As instruções `AuthType`, `AuthName`, `AuthUserFile` e `AuthGroupFile` estão relacionados à autenticação. `AuthType` especifica o tipo de autenticação. `Basic` é a mais comum e única que permite controle total por parte do autor do site. `AuthName` informa um nome (qualquer) para este recurso de autenticação (é obrigatório para que se possa usar as instruções seguintes).

[1] *Server side includes* – veja capítulo sobre este assunto mais adiante.

`AuthUserFile` informa a localização do arquivo onde estão os usuários e senhas. No tipo de autenticação `Basic`, este arquivo precisa ser criado usando a aplicação de linha de comando `htpasswd` (disponível no *Apache Unix*) da forma:

```
$ htpasswd -c /caminho/ate/arquivo/.senhas usuario
```

O programa pedirá que se digite e se confirme a senha digitada. Rodar `htpasswd` sem argumentos mostra uma tela com a sintaxe e lista de opções disponíveis. A opção `-c` (*create file*) indica que o arquivo deve ser criado. Para criar novos usuários em arquivo já existente, deve-se eliminar o "`c`":

```
$ htpasswd /caminho/ate/arquivo/.senhas outroUsuario
```

Esse programa pode ser chamado por um programa CGI (iniciado pelo servidor Web) de forma que novos usuários possam se cadastrar on-line. Ele automaticamente criptografa as senhas. No *Apache Windows* o programa também está disponível (como aplicação MS-DOS `htpasswd.exe`):

```
c:\Apache\bin\htpasswd -c c:\Apache\auth\usuarios.txt helder
```

Mas é possível simplesmente criar um arquivo de texto comum, já que o *Windows* não criptografa as senhas. O arquivo deve conter um nome e uma senha por linha da forma:

```
morpheus:lcfr666:
malacoda:mlbolge:
dracula:sangue:
lestat:abcd123:
```

A instrução `AuthGroupFile` informa os usuários (que devem existir no arquivo de senhas) que pertencem a determinado grupo. Cada linha do arquivo informa o grupo, seguindo por dois pontos e uma lista de usuários:

```
vampiros: dracula lestat ivanovich ...
diabos: malacoda draghinazzo rubicante farfarello barbariccia
```

As diretivas `<Limit>` estabelecem limites para os métodos do browser. `GET` é o método usado para buscar páginas, imagens, programas, etc. `POST` só é usado em formulários. No exemplo acima, as restrições para o método `GET` (ou seja, quando o browser tentar pegar qualquer coisa no diretório) são:

```
require user morpheus
require user malacoda
require group vampiros
deny from .batcaverna.com
```

As instruções `require user` informam os usuários que terão acesso a esta área do site. A instrução `require group` informa que o grupo dos `vampiros` (todos os usuários que pertencem ao grupo) também terá acesso. A instrução `deny from .batcaverna.com` indica que não terão acesso usuários que estiverem navegando a partir do domínio `.batcaverna.com` (`batman.batcaverna.com` e `robin.batcaverna.com` não terão acesso!). Todos os que tiverem o acesso negado serão desviados para uma página padrão, estabelecida na configuração do servidor, que informará que a autenticação falhou.

Se o limite para `POST` não for definido, ele terá os parâmetros *default* declarados no arquivo `access.conf` (ou `httpd.conf`). Geralmente só quem recebe `POST` são programas CGI. Supondo que eles existam nesse diretório, apenas os usuários de `.vampiros.org` teriam condições de usá-los. As restrições para `POST`, neste arquivo `.htaccess`, são:

```
order allow, deny
allow from .vampiros.org
deny from all
```

`Order` informa a ordem em que serão processadas as instruções seguintes. Se a ordem for `deny, allow`, ninguém terá acesso a este diretório via `POST`. Com `allow, deny` o servidor lê a linha `allow from` que permite acesso aos usuários de `castelo.vampiros.org` e outras máquinas desse domínio enquanto nega para todos os outros (`deny from all`).

Configuração: arquivo de acesso

Se você estiver usando o Apache sob *Windows95* (e não 98, NT ou 2000), poderá ser impossível criar um arquivo começando em "`.`" como `.htaccess`. Apesar disso, você ainda pode configurar o acesso por diretório desde que informe, na configuração do Apache, outro nome de arquivo padrão que será consultado por ele ao entrar em um diretório. O nome do arquivo de acesso por diretório é alterado através da propriedade **`AccessFileName`**:

```
AccessFileName .htaccess
```

Mude o nome para um arquivo que possa ser criado no Windows, por exemplo, htaccess.txt, e reinicie o servidor:

```
AccessFileName htaccess.txt
```

Com essa alteração, o servidor agora procurará sempre por um arquivo chamado htaccess.txt ao entrar em um diretório. Se o encontrar, seguirá suas instruções que poderão restringir ou ampliar o acesso a partir daquela parte do site. Uma vez que você conseguir autenticar-se em um diretório, você terá livre acesso a ele. A janela que pede o nome e a senha não aparecerão mais, mesmo que você aperte o botão *Atualizar* (*Reload*) várias vezes. Ela só voltará se você fechar o browser e abri-lo novamente.

Estas últimas seções apresentaram uma pequena introdução ao controle de acesso usando o `.htaccess`, arquivo de configuração de acesso do servidor *Apache* – o servidor que atualmente serve mais da metade das páginas da Internet. Esse conhecimento permite que o Web designer tenha o poder de controlar níveis de acesso do site, sem que ele tenha que programar ou ter permissões de acesso aos arquivos de configuração do servidor Web. O controle, para a maior parte dos sites, requer arquivos menores e mais simples que os mostrados.

3.4. Instalação e configuração básica de servidores Microsoft

O segundo servidor mais usado no mundo é nativo dos sistemas Windows. O *Internet Information Server* (IIS) está disponível em sistemas *Windows NT Server*. Nos sistemas *Windows NT Workstation, Windows 98* e *2000* pode-se opcionalmente instalar os servidores *Personal Web Server* ou *Peer Web Server* (PWS) que se assemelham ao IIS mas oferecem uma interface mais simples, com menos recursos configuráveis.

A interface de administração do PWS e IIS é gráfica e pode ser feita através de uma aplicação do sistema operacional ou através da Web (páginas HTML), permitindo a administração remota. Nesta seção apresentaremos a configuração básica do *Personal Web Server* (PWS), distribuído com o Windows98.

Embora o Windows ofereça uma interface gráfica para a administração do servidor, a mudança de parâmetros não disponíveis através de sua interface geralmente requer a edição de arquivos, associações e chaves do Registro do Windows, cuja edição é complexa até para programadores experientes. O *Internet Information Server* permite alterar a maior parte dos parâmetros de interesse do administrador do site sem precisar recorrer ao *Registro*. No caso do *Personal Web Server*, que foi projetado para uso em computadores desktop, não há como mudar a porta de serviços (número 80) sem editar o *Registro*.

Neste curso, provavelmente você está usando uma máquina *Windows98* ou *NT Workstation* que não tem o IIS, portanto, teremos que trabalhar com o PWS. Para ter o *Apache* e o PWS rodando na mesma máquina ao mesmo tempo, é preciso que cada um deles utilize uma porta diferente. Como é mais fácil mudar a porta do *Apache* (é só alterar a propriedade Port do arquivo `httpd.conf`), mude-a para 8080 (ou outro número fácil de lembrar, maior que 1024). Quando os dois servidores estiverem rodando, será possível ter acesso a dois sites na mesma máquina, informando a porta na URL do *Apache*:

- `http://paquistao.ibpinet.com.br/` Esta é a URL do PWS
- `http://paquistao.ibpinet.com.br:8080/` Esta é a do Apache na porta 8080

A instalação do PWS pede apenas para informar a localização do diretório onde ficarão armazenados os arquivos servidos pelo servidor. Normalmente esses arquivos ficam em `c:\Inetpub\` (raiz do servidor equivalente à `ServerRoot` do *Apache*) e em `c:\Inetpub\wwwroot\` fica a raiz de documentos (`DocumentRoot` no *Apache*).

Publicação de páginas

O PWS oferece um assistente de publicação simples que automaticamente localiza seus arquivos e os transfere para a raiz de documentos. Isto também é realizado pelo *FrontPage*, que utiliza o PWS como servidor auxiliar.

Para publicar as páginas "manualmente", basta localizar o diretório raiz de documentos (geralmente `c:\Inetpub\wwwroot\`) e copiar os arquivos para ele. Novos diretórios criados abaixo da raiz automaticamente ficam disponíveis para visualização remota.

Instalação de programas

Para que um servidor possa rodar programas por solicitação do browser, eles precisam ser instalados no servidor. A instalação de programas em servidores PWS é simples. Basta copiá-los para o diretório `c:\Inetpub\scripts\` ou para um diretório qualquer que tenha permissões de execução. As permissões podem ser mudadas através de uma janela de diálogo.

Controle de Acesso

Esta opção só está disponível nos servidores IIS (e não no PWS, disponível no laboratório). Para maiores informações sobre os recursos do PWS, consulte a ajuda on-line do servidor.

3.5. Exercícios

1. Você pode simular um browser através da aplicação Telnet. Configure o seu cliente Telnet (Preferências) para que ele mostre o *eco local* dos comandos digitados. Conecte-se, então à porta 80 (ou 8080) de uma máquina que possua servidor Web:

   ```
   telnet www.ibpinet.net 80
   ```

   Escreva o comando a seguir (e não use *backspace* para corrigir), com o `GET` em maiúsculas:

   ```
   GET /book/ch1.html HTTP/1.0
   ```

   Aperte Enter duas vezes. O servidor receberá sua requisição e pensará que você é um cliente HTTP. Ele devolverá um arquivo HTML e depois encerrará a sua conexão.

2. Instale o servidor Apache na sua máquina do laboratório. Inicie o servidor e teste se está tudo OK carregando a sua página principal. Identifique seu diretório raiz e altere a página de índice que é carregada na raiz do servidor.
3. Mude a porta do servidor Apache para 8080 (no arquivo httpd.conf). Reinicie o servidor e teste seu funcionamento mais uma vez. Qual é a nova URL do servidor?
4. Acrescente outras opções de documento índice no seu Apache como index.htm e default.htm. Reinicie o servidor e teste o funcionamento.
5. Crie um mapeamento (Alias) para que o diretório onde você tem armazenado suas páginas HTML seja visível através do servidor Web. Você precisará inventar um complemento da URL (`/curso/` por exemplo) e associar esse complemento ao diretório do Windows onde estão seus arquivos. Reinicie o servidor e teste o funcionamento.
6. Interrompa temporariamente o servidor e abra seus arquivos de controle (`logs`) de erro e de acesso. Identifique as conexões que você fez recentemente.
7. Limite o acesso a determinadas áreas do seu site criando um arquivo `.htaccess`. Crie um arquivo de senhas que contenha apenas o seu nome e uma senha (no formato aceito pelo `.htaccess`). O `.htaccess` deve apontar para ele. Use qualquer nome como `AuthName`.
8. Repita o exercício anterior na sua conta *Unix*. Você precisará, neste caso, criar um arquivo de senhas com o programa `htpasswd`.

4. Aplicações Web

4.1. O que são aplicações Web

Aplicações Web são aplicações cujo ambiente de execução é a plataforma Web. Suas funções e recursos podem estar distribuídos por várias localidades, e disponibilizados através de um ou mais *servidores* HTTP. A interface do usuário de uma aplicação Web é construída por um browser (ou outro *cliente* HTTP), após interpretar uma página escrita em HTML, fornecida pelo servidor.

As aplicações Web às quais nos referimos são as que são capazes de realizar tarefas mais sofisticadas que o simples acesso a informações em hipertexto. Permitem a construção de sites interativos. HTML não possui recursos de programação nem recursos de apresentação visual, portanto, não é suficiente para o desenvolvimento dessas aplicações Web. A linguagem HTML apenas fornece instruções de *marcação* de texto que são utilizadas pelo browser para:

- formatar uma página de informação (usando uma determinada *folha de estilos*, geralmente definida pelo browser), com texto, tabelas e componentes de formulário como botões, caixas de texto, etc.
- vincular imagens à página, formatando-as juntamente com o texto (se possível),
- vincular recursos multimídia como som, vídeo, applets e *plug-ins*, e
- habilitar eventos em vínculos de hipertexto e botões para permitir que o browser carregue outras páginas ou recursos cujos endereços estão embutidos no código.

Com HTML *apenas*, a única aplicação Web que se pode construir é aquela que permite a navegação dentro de um banco de informações em hipertexto. Esta aplicação é básica e fornecida por qualquer browser.

Para ir além da navegação, é preciso estender as capacidades básicas do HTML através outras tecnologias que estendem as capacidades de um cliente ou servidor. Essas extensões podem ser classificadas em relação à sua localização dentro da arquitetura cliente-servidor. Recursos *lado-cliente* executam no navegador, e só dependem dele para funcionar. Recursos *lado-servidor* executam no servidor e só precisam do suporte do servidor. Um browser, porém, pode iniciar uma aplicação que executará no servidor e o servidor poderá enviar páginas ou componentes que só serão interpretados ou executados no browser. Freqüentemente, uma aplicação Web utiliza tanto recursos *lado-cliente* quanto *lado-servidor*.

Aplicações Web com resursos lado-cliente

Existem dois tipos de extensões *lado-cliente*: os *componentes* ou *objetos*, que funcionam como extensões *executadas* como se fossem parte do browser; e os *scripts* (roteiros), que são *interpretados* pelo browser juntamente com o HTML.

Os componentes, como os Java *Applets*, controles *ActiveX* ou *plug-ins* de som, vídeo, Flash VRML e outros, podem acrescentar novos recursos ao browser, permitir que o mesmo suporte outros formatos de multimídia, ou realizar tarefas bem específicas. São aplicações completas e geralmente interagem pouco com a página HTML,

utilizando-a somente para obter um contexto gráfico para exibir sua própria interface. Componentes são objetos externos à página, e, como qualquer objeto independente do texto, são carregados através de uma requisição à parte (como é feito com as imagens).

Os *scripts* estendem a linguagem HTML (e não o browser). Geralmente são embutidos dentro do próprio código HTML. São *interpretados* enquanto o browser carrega a página. O próprio código HTML é um *script* que é interpretado pelo browser para definir a estrutura da página. Um possível bloco CSS (*Cascading Style Sheets*) embutido no HTML é outro *script* que estende o código HTML para definir a apresentação e *layout* de uma página. Estruturas de programação podem ser embutidas em uma página usando JavaScript, que introduz no HTML a capacidade de manipular eventos, realizar controle de fluxo de um programa, suporte a operações matemáticas, acesso a variáveis do browser entre outras possibilidades.

Enquanto as tecnologias *lado-cliente* são ótimas para realizar operações locais como validação, geração de gráficos, etc. não servem para a maior parte das operações que exigem persistência de dados. Operações de acesso a rede que utilizam disco local geralmente são desabilitadas ou bastante restritas no cliente, por questões de segurança. Também podem ser pouco eficientes. Nesses casos, a melhor alternativa é apelar às tecnologias *lado-servidor*.

Aplicações Web com recursos lado-servidor

Existem várias arquiteturas diferentes que implementam suporte a extensões em servidores Web. A mais popular delas é a tecnologia CGI – *Common Gateway Protocol*, uma especificação que permite o desenvolvimento de aplicações *gateway* que servem como ponte para que o browser possa realizar tarefas através do servidor. CGI não é linguagem. É apenas uma especificação que dita regras para que programas intermediados pelo servidor Web possam ser implementados usando *qualquer* linguagem. Aplicações CGI podem ter sua execução solicitada por uma requisição do browser e podem servir de ponte para qualquer aplicação ou dados localizados no servidor.

CGI já foi a única forma de interatividade via Web. Hoje, a cada dia, vem sendo substituída por soluções mais eficientes, muitas delas proprietárias. As tecnologias mais comuns, que diferentemente do CGI, dependem do tipo e plataforma do servidor são o ISAPI, da Microsoft, e o NSAPI, da Netscape. Estas tecnologias oferecem módulos ou "plug-ins" que permitem que um programador desenvolva extensões destinadas ao tratamento de dados e comunicação pelo servidor, podendo substituir totalmente o CGI, com ganhos de desempenho, porém com um razoável acréscimo de complexidade e perda de portabilidade.

Entre as APIs e o CGI, porém, existem os *componentes* para servidor. São programas em Java (servlets) ou objetos *ActiveX* que fazem o mesmo que as APIs, porém mantendo uma portabilidade maior. Mais simples ainda são os *scripts* ou roteiros de código, embutidos em páginas Web, que servidores devidamente configurados usam para realizar tarefas de transformação de dados e comunicação com outros programas. Com estas tecnologias, o conteúdo da página pode ser alterado no próprio servidor no momento do envio, através da interpretação de *scripts*, que também servem de *gateway* com aplicações no servidor, como bancos de dados. Todas estas tecnologias substituem completamente o CGI e são geralmente, mais eficientes. Todas, também, operam sobre o servidor HTTP, da mesma forma que CGI. Conhecer CGI, portanto, é bastante útil para o domínio de qualquer uma dessas tecnologias.

5. Instalação do Serviço de Aplicações

5.1. Programas CGI

Como foi mencionado antes, CGI, ou *Common Gateway Interface* é um mecanismo que permite que browsers e outros clientes Web executem programas em um servidor Web. Programas que usam esse mecanismo são chamados de programas CGI e são largamente utilizados para produzir páginas dinâmicas, para processar conteúdo de formulários, acessar bancos de dados entre outras aplicações.

O *programa CGI* é um programa que recebe dados em um formato pré-definido e que geralmente produz como saída uma página HTML, imagem ou outras informações utilizáveis pelo cliente Web. Quem implementa o CGI é o servidor Web e a forma como faz é baseada em padrões do W3C.

Programas CGI podem ser escritos em qualquer linguagem de programação. O seu funcionamento é totalmente baseado nos dados que recebe e nos resultados que fornece. São como *caixas-pretas* com uma entrada e uma saída. Não interessa o conteúdo da caixa-preta (a linguagem). Tanto faz usar C, Perl, Java, Fortran, Basic, Delphi ou até uma linguagem proprietária para realizar a tarefa solicitada. O que interessa é que o programa saiba ler os dados recebidos pelo browser através da entrada padrão (porta do servidor) e variáveis de ambiente e tenha a capacidade de escrever sua resposta em um formato legível pelo cliente Web (em HTML, em GIF, em JPEG, etc.). Programas CGI geralmente são escritos nas linguagens mais populares e fáceis de usar para um dado sistema operacional. É comum utilizar linguagens de roteiro interpretadas (Shell, DOS, Perl, AppleScript) para rotinas pequenas e simples e linguagens compiladas para trabalhos mais complexos e permanentes (VB, C, C++, Delphi).

A escolha da linguagem depende da disponibilidade e do que se pretende fazer. Perl tem se tornado um padrão de mercado por estar disponível em quase todas as plataformas que possuem servidores Web. Serve para aplicações simples e complexas. C e C++ têm sido usadas em aplicações que exigem maior robustez e desempenho. Mas, como o que interessa para o servidor Web é a interface de entrada e saída da aplicação, basta que a linguagem escolhida possua recursos que facilitem o acesso a dados via variáveis de ambiente e seja capaz de escrever dados em formatos de 8 bits.

Portanto, se você sabe programar, não precisa aprender Perl. Pode usar a sua linguagem favorita para desenvolver aplicações CGI. Neste curso, usaremos Perl para demonstrar exemplos de CGI mais adiante.

5.2. Implantação do CGI

Para que o servidor decida tentar executar um objeto solicitado pelo browser em vez de enviá-lo à saída padrão (para *download*) é necessário configurá-lo para suportar CGI.

O comando GET simplesmente faz com que o servidor retorne o objeto requisitado ao browser. Se o link de GET for para um programa chamado "contador.exe" em um servidor Windows, por exemplo, ele irá enviar o programa para o browser, que, por sua vez, apresentará uma janela ao usuário perguntando se ele deseja fazer o *download* do programa. Mas se o servidor estiver configurado de tal forma a identificar "contador.exe" como sendo um *programa CGI*, ele tentará *executar* o programa e, caso tenha sucesso, enviará para a saída padrão a informação gerada na saída do mesmo.

A forma de configurar CGI varia de servidor para servidor. Na maioria dos servidores pode-se definir um programa CGI como sendo qualquer executável que esteja em um diretório especial, configurado para permitir a execução de programas CGI. Pode-se também definir CGI como sendo um tipo especial de dados, identificado pela extensão do nome do arquivo.

Servidores Unix

Para definir uma área de programas CGI em servidores HTTP tradicionais (CERN, NCSA, Apache) é necessário modificar arquivos de configuração. No *Apache*, a modificação pode ser feita no arquivo `srm.conf` (ou `httpd.conf`, dependendo da instalação) usando a propriedade **`ScriptAlias`**:.

```
ScriptAlias    /cgi-bin/    /dev/lib/httpd/cgi-bin/
```

A linha acima faz com que a URL relativa `/cgi-bin/` represente, para o servidor Web, o diretório `/dev/lib/httpd/cgi-bin/` que conterá programas que serão executados pelo servidor Web, quando o browser os requisitar. No *Apache Windows* a regra é semelhante:

```
ScriptAlias    /cgi-bin/    "c:\programas\cgi\"
```

Não é preciso definir um *diretório CGI* para que o servidor rode programas CGI. O importante é que o servidor saiba quando deve rodar um programa e quando deve devolvê-lo ao browser para *download*. Uma outra forma de configurar CGI é atribuir uma extensão de arquivo especial aos programas executáveis, definindo CGI como *tipo de dados* identificado por extensão de nome de arquivo. Assim configurados, programas CGI podem estar localizados em qualquer lugar, mesmo em diretórios de documentos.

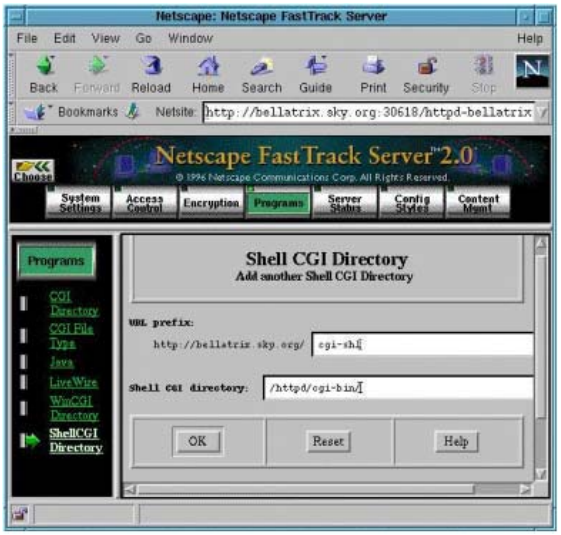
Para definir CGI como um *tipo de dados* no servidor *Apache*, acrescente (ou remova o comentário) no `s-rm.conf` (ou `httpd.conf`) a linha:

```
AddHandler  cgi-script  .cgi
```

A extensão escolhida não precisa necessariamente ser `.cgi`. Você pode escolher outra. Deve ser uma extensão que não seja comum no seu sistema ou que tenha algum significado especial fora do servidor Web. Um grave furo de segurança é definir, em um servidor Windows, uma linha como:

```
AddHandler  cgi-script  .exe
```

Isto fará que um programa executável sempre seja executado pelo servidor Web. Não será mais possível fazer download de tais programas, pois, mesmo localizados em diretórios de documentos eles seriam executados. É fácil perder o controle em situações assim, portanto, não defina .exe como extensão no Windows nem outras extensões críticas, como `.pl` em qualquer sistema que rode Perl.



Nos outros servidores *Unix* (*Netscape FastTrack* e *Enterprise Server*) a configuração de CGI (por diretório ou tipo de arquivo) tanto pode ser realizada *manualmente* (editando os arquivos `obj.conf` e

`magnus.conf` do servidor) como graficamente, usando a sua interface via Web (imagem ao lado).

Servidores Windows

Nos sistemas *Windows* existem dois (ou três, dependendo do servidor) tipos diferentes de CGI. A maior parte dos servidores não é capaz de distinguir entre eles automaticamente e é necessário configurar o CGI para o tipo correto a ser usado.

O primeiro tipo é o *Windows CGI*, que suporta programas que rodam sob o ambiente gráfico Windows (e não rodam sob MS-DOS).

O *DOS CGI* é também chamado simplesmente de CGI e suporta programas que rodam através do *MS-DOS* como arquivos de lote (`.BAT`), e executáveis DOS (`.COM`, `.EXE`).

Finalmente, alguns servidores distinguem ainda, entre os programas CGI que rodam sob o MS-DOS, o chamado *Shell CGI*, que consiste dos programas que requerem um interpretador para rodar. O desenho ao lado mostra a configuração desse tipo de CGI em um servidor Netscape. Exemplos são programas escritos em Perl, Java ou Basic interpretado.

Nos servidores IIS e PWS (Microsoft) qualquer diretório configurado como executável, acessível dentro de uma Web local, terá permissão de executar DOS CGI ou Windows CGI. A instalação de programas que requerem CGI interpretado em Perl é mais complicada, e exige a intervenção no Registro do Windows (PWS 4.0).

5.3. Alternativas ao CGI

CGI é uma tecnologia portátil, flexível e suportada por praticamente todos os servidores Web. Há, porém, diversas limitações na tecnologia CGI que podem até inviabilizar certos tipos de aplicação.

Para cada *requisição* de um cliente, um novo *processo* é criado, mesmo que seja para executar a mesma aplicação e fazer as mesmas coisas. Quando raramente há mais de um cliente conectado, ou quando a requisição não ocupa tempo do servidor, isto pode não ser um problema. Porém quando há muitos clientes realizando tarefas demoradas, o desempenho do servidor poderá cair drasticamente.

Programas CGI, como são externos ao servidor, não podem aproveitar recursos ou extensões disponíveis no servidor. Também não conseguem refletir o estado do servidor e saber se este mudou depois que o programa foi executado.

Para eliminar ou atenuar essas limitações, têm surgido várias alternativas ao CGI. Todas trazem maior integração com o servidor, o que melhora significativamente o desempenho. Por outro lado, essa maior integração já não permite que os programas sejam tão independentes como eram com CGI. Não há mais a liberdade de se escolher a linguagem de programação. É preciso ainda programar usando bibliotecas específicas, fornecidas pelo fabricante da tecnologia ou do servidor, que muitas vezes são proprietárias, limitando-se a determinado fabricante.

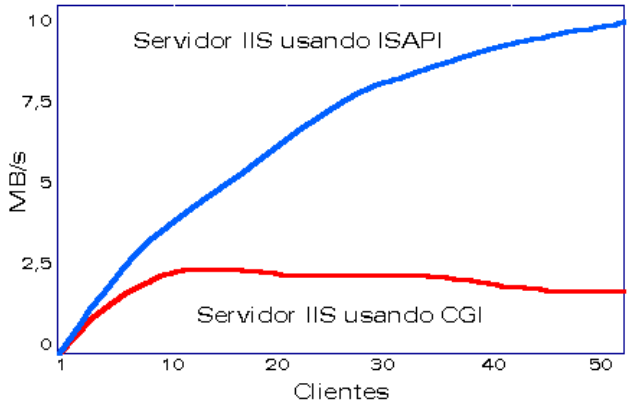
As soluções mais eficientes são as chamadas APIs de servidor (*Server APIs*). Uma API é uma coleção de módulos (funções, classes, objetos) que um programador pode usar para desenvolver aplicações aproveitando estruturas padrão previamente desenvolvidas para que não precise construir todos os programas do nada. Uma API de servidor expõe a estrutura de um servidor através de uma linguagem de programação, definindo funções que um programador pode usar e outras que ele precisa redefinir. Com esses recursos, ele pode desenvolver programas que se integrarão ao servidor e poderão substituir CGI de forma até mais simples, desde que o programador conheça a linguagem na qual foi desenvolvida a API.

A maior parte das SAPIs são proprietárias e exigem conhecimento de linguagens como C++. As mais populares, como ISAPI (da Microsoft) e NSAPI (da Netscape) dispõem de APIs em outras linguagens populares como Delphi, Visual Basic e Perl.

O desempenho de um servidor que usa uma SAPI em geral é muito melhor que um servidor que usa CGI, mesmo que os programas sejam escritos na mesma linguagem. Quando há mais de um cliente simultâneo no

servidor, a diferença aumenta mais ainda. Veja o gráfico abaixo comparando o desempenho médio de programas equivalentes usando CGI e ISAPI em um servidor Microsoft *Internet Information Server* rodando em Windows NT.

As SAPI não substituem totalmente o CGI por não terem a mesma portabilidade, serem dependentes de fabricante e exigirem a compilação de programas grandes mesmo quando a tarefa a ser realizada é simples e exige poucas linhas de código se feita em CGI. Módulos SAPI se integram tão em ao servidor que passam a fazer parte dele (no Windows, são arquivos *.dll* que rodam junto com o servidor). Isto pode trazer desvantagens. Um *bug* no programa, por exemplo, pode derrubar todo o servidor, impedindo que ele continue servindo páginas.



Fonte: PCMagazine Labs

Hoje existem várias soluções baseadas em SAPIs que resolvem parte desses problemas. Um SAPI independente de plataforma e de fabricante que é capaz de lidar com seus erros sem derrubar o servidor é o Java Servlet API. Os módulos são desenvolvidos em Java e se chamam *servlets*. Podem substituir CGI, os módulos SAPI proprietários e são suportados pela maioria dos servidores de forma nativa ou através de *plug-in*.

Mas desenvolver SAPIs é um trabalho de programação. Muitas vezes o trabalho de resposta a um formulário consiste apenas em devolver uma página preenchida com informações de um banco de dados, de uma caixa postal, ou de duas ou três linhas alteradas de acordo com preferências do usuário. Para desenvolver tais páginas de resposta com SAPIs, seria preciso saber programar na API para imprimir principalmente linhas de HTML. No fim, o programador acaba fazendo um trabalho que o Web designer devia estar fazendo, mas não faz porque precisa conhecer uma linguagem complexa como Java ou C++.

A solução para esse problema surgiu com os roteiros ou *scripts* do servidor. Enquanto os programas CGI e as SAPIs precisam embutir o código HTML dentro de suas instruções de programa, os *scripts* fazem exatamente o contrário: embutem trechos de programa dentro de páginas HTML. Embora ainda exista programação, a criação de páginas de resposta fica muito mais simples. Um Web designer pode recortar e colar os trechos de código destinados ao servidor e se concentrar no visual da página. As linguagens utilizadas podem ser desde Java até linguagens mais simples como JavaScript e VBScript.

A primeira solução usando *scripts* foi o LiveWire, da Netscape. Depois veio o *Cold Fusion*, da Allaire. Hoje as tecnologias mais populares são o ASP – *Active Server Pages* da Microsoft, o JSP – *Java Server Pages* da Sun e o PhP, que mistura ASP, Java e Perl em uma linguagem que tem se tornado popular entre os desenvolvedores de aplicações Web.

Embora os *scripts* sejam embutidos em páginas HTML, eles jamais chegam até o cliente. Páginas que possuem *scripts* embutidos para o servidor geralmente têm uma extensão de nome de arquivo diferente para que o servidor saiba que não se trata de uma página HTML qualquer. Ao receber uma requisição para enviar tal arquivo para o cliente, o servidor o abre e analisa o código HTML à procura de códigos especiais que deve interpretar. Esses códigos geralmente estão entre símbolos como `<% ... %>` ou `<? ... ?>`. A página é então tratada como um programa. Tudo o que for HTML é copiado para a saída padrão (onde o browser pode pegar a informação). O que não for HTML é interpretado, gerando, no final, código HTML dinâmico que não estava presente no arquivo original. As instruções podem fazer qualquer coisa que um programa CGI pode fazer, com desempenho próximo ao das SAPIs. O browser sempre recebe uma página 100% HTML (contendo possivelmente, código *JavaScript* para o cliente).

São, portanto, várias as opções para desenvolvimento de aplicações do lado do servidor. Qual usar? Depende de vários fatores: disponibilidade, segurança, velocidade, custo, simplicidade. Se o objetivo é fazer um

sistema simples, tipo uma busca leve, um *guestbook*, formulário de *feedback*, etc., talvez não valha a pena investir tempo em soluções muito complicadas, podendo-se aproveitar o oceano de recursos CGI disponíveis na rede. Por outro lado, se o desempenho é um fator crítico, convém analisar outras alternativas como os SAPI. Se a resposta do servidor freqüentemente consiste na geração de páginas, os *scripts* de servidor podem ser a melhor solução, pois podem também integrar-se com SAPIs e realizar tarefas complexas.

CGI continua como base para estas tecnologias. A interface CGI lida com detalhes da programação do servidor em *baixo-nível*, se preocupando com o formato exato de cabeçalhos, requisições, e outros detalhes do protocolo HTTP. Conhecendo-se CGI, a escolha e uso de uma outra tecnologia torna-se muito mais fácil, pois todas utilizam os mesmos conceitos, embora ofereçam funções e métodos que permitem a programação em um nível mais alto. Aprender a usar CGI, portanto, será útil mesmo se você decidir usar outra tecnologia mais sofisticada.

Os capítulos seguintes abordarão a interface CGI em maiores detalhes e com exercícios práticos. O foco será na especificação CGI e não na programação. Usaremos a linguagem Perl para demonstrar algumas aplicações, mas evitaremos entrar em detalhes quando aos detalhes dessa linguagem.

5.4. Exercícios

1. Configure um diretório CGI no servidor Apache para que ele seja capaz de rodar programas.
2. Na sua máquina deve existir o ambiente *Perl* instalado. Deve haver um diretório `D:\Perl` ou `C:\Perl` e, se você abrir uma janela de MS-DOS, digitar

   ```
   perl -V
   ```

   deverá ver uma tela informando a versão (deve ser 5.0005) e o fabricante (*ActiveState*) do *Perl* instalado. Se tudo estiver funcionando, crie o seguinte programa usando um editor de textos qualquer (como *WinEdit* ou o *Bloco de Notas* do *Windows*). Chame-o de `teste.pl`. A primeira linha poderá conter um c: ou um d: dependendo de onde está instalado o seu *Perl*:

   ```
   #!/c:\Perl\bin\perl.exe
   print "Content-type: text/html\n\n";
   print "<h1>Olá Mundo! Eu sou um programa em Perl!</h1>";
   print "<p>O seu PATH é: " . $ENV{'PATH'};
   ```

   Digite o programa *exatamente* como está acima (com exceção, talvez, do drive `c:` ou `d:`). Não troque maiúsculas por minúsculas e não esqueça as aspas, os pontos, os apóstrofes e os pontos e vírgula. Salve o programa no diretório que você configurou para rodar CGI.
   Rode o seu programa em linha de comando. Ele deverá imprimir exatamente o texto a seguir (com exceção talvez do valor do `PATH`):

   ```
   Content-type: text/html

   <h1>Olá Mundo! Eu sou um programa em Perl!</h1>
   <p>O seu PATH é: c:\Windows\Command;c:\Perl\bin;c:\Windows;.
   ```

   Observe que a segunda linha está em branco. Isto é importante. Abra agora o seu browser e digite uma URL que aponte para o programa que você acabou de criar (através do diretório virtual que você definiu no `ScriptAlias`). Se a mensagem `Olá Mundo` aparecer, a sua instalação de CGI e o seu programa foram um sucesso!
3. O seu ambiente CGI está configurado no *Unix* abaixo do caminho

   `/i1/paginas/cgi/wd/turmax/seulogin` (para a turma X)

No Apache do *Linux*, há uma declaração dizendo:

```
ScriptAlias /wd/turmax/seulogin/cgi-bin/
            /i1/paginas/cgi/wdx/seulogin
```

Crie um roteiro *Shell* que imprima a mesma coisa que o programa em *Perl* do exercício anterior. Chame-o de `teste.sh`. Na primeira linha você deve colocar `#!/bin/sh` para identificar o interpretador (que não é *Perl*)
Transfira o programa para a sua conta (via FTP) e depois copie-o para o seu diretório CGI no *Linux* (via *Telnet*). Habilite a execução do programa (mudando a permissão) e rode-o em linha de comando (para ver se está tudo OK). Depois, aponte seu browser para a URL que solicita o programa, para testá-lo. O resultado deverá ser o mesmo alcançado no exercício anterior.

5.5. Testes

1. Marque as afirmações falsas.
   a) É preciso que um browser ofereça suporte a JavaScript para entender páginas, armazenadas no servidor, que contêm código ASP ou PHP, mesmo que o servidor Web suporte estas tecnologias.
   b) Para carregar uma página que possui 5 imagens e uma applet Java, o browser precisa fazer pelo menos 7 requisições independentes ao servidor.
   c) O código HTML dos arquivos armazenados no servidor é sempre analisado e interpretado pelo servidor Web antes de ser enviado para o browser.
   d) É preciso que o servidor Web suporte Java e JavaScript do lado do servidor (ASP por exemplo) para que possa servir páginas HTML com JavaScript e applets Java aos seus clientes.
   e) É possível implementar um contador de acessos – que conta o número de vezes que uma determinada página foi acessada – usando *apenas* técnicas de interatividade no cliente como applets Java ou JavaScript.

2. Marque as afirmações verdadeiras.
   a) Programas que serão executados pelo servidor usando a tecnologia CGI podem ser escritos em qualquer linguagem de programação, não apenas Perl.
   b) Programas CGI sempre têm seu código interpretado pelo servidor Web.
   c) Para cada cliente que usa uma aplicação Web baseada em CGI o servidor precisará iniciar um novo processo de forma que se há 100 clientes conectados a uma aplicação de banco de dados, há pelo menos 100 aplicações rodando cuja execução foi iniciada pelo servidor Web.
   d) A única forma de fazer um servidor Web se comunicar com um banco de dados ou outra aplicação no cliente é através da interface CGI, ou seja, é preciso que um programa CGI faça a intermediação entre a aplicação e o servidor Web.
   e) O servidor poderá alterar o conteúdo de uma página armazenada no seu disco antes de enviá-la para o browser.

3. Um programa, ao ser executado em linha de comando (*Unix Shell* ou *MSDOS Prompt*) imprime *exatamente* o texto a seguir.

```
Content-type: text/html
<HTML>
<HEAD><TITLE>Resposta CGI</TITLE>
</HEAD>
<BODY>
<P>Esta é uma resposta CGI.
</BODY></HTML>
```

Ele é devidamente configurado para funcionar como programa CGI em um servidor Web. Tem permissão de execução e é colocado em diretório *CGI-BIN* onde o servidor já executou vários outros programas lá ar-

mazenados, com sucesso. Ao ser solicitado por um browser, no entanto, o servidor retorna "**500 Internal Error**", sinalizando que algo está errado. O que está errado? Marque uma alternativa.

a) Falta acrescentar outros cabeçalhos como "Date" e "Server".
b) Falta acrescentar o cabeçalho "Location", apenas.
c) Falta terminar o cabeçalho com uma linha em branco.
d) O problema pode ser do browser, que não suporta CGI.
e) A saída do CGI está correta. O problema certamente é outro.

4. Considere o seguinte formulário HTML:

```
<form action="/cgi-bin/query.sh" method="POST">
<input type=text name="keyword">
<input type=Submit value="Submeter Pesquisa">
</form>
```

O programa CGI query.sh é escrito na linguagem Bourne-Shell e contém o seguinte código:

```
#!/bin/sh
echo Content-type: text/html
echo
echo "<html><p>"
echo $QUERY_STRING
echo "</p></html>"
```

Nesta linguagem, "`echo`" é uma instrução que imprime o texto que a segue até o fim da linha (inclusive o caractere de nova-linha). Texto entre aspas é impresso sem as aspas. Palavras precedidas por "`$`" são variáveis de ambiente. O CGI funciona sem problemas e o servidor não retorna erros. O que aparece escrito no browser, depois que um usuário escreve a palavra `Leão` e aperta o botão "Submeter Pesquisa"?

a) o texto `keyword=Leão`
b) nada
c) o texto `keyword=Le%E3o`
d) o texto
```
<html><p>
keyword=Le%E3o
</p></html>
```
e) uma caixa de texto vazia, e um botão com o rótulo "Submeter Pesquisa".

Criação de Web Sites II

2

Formulários HTTP e CGI

Conteúdo

6. Formulários 65
6.1. Elemento <FORM> 65
Botões de envio 66
Entrada e envio de dados 66
Múltiplos campos de entrada de dados 67
Outros elementos de entrada de dados 68
6.2. Elemento <INPUT> 68
6.3. Botões (TYPE=BUTTON, SUBMIT, RESET ou IMAGE) 69
6.4. Campos de texto de uma linha (TYPE='TEXT ou PASSWORD) 70
6.5. Campos ocultos (TYPE=HIDDEN) 70
6.6. Chaves booleanas do tipo "botão de rádio" (TYPE=RADIO) 71
6.7. Elemento <LABEL> 71
6.8. Chaves booleanas tipo "caixas de checagem" (TYPE=CHECKBOX) 72
6.9. Upload de arquivos (TYPE=FILE) 73
6.10. Elemento <TEXTAREA> 73
6.11. Elementos <SELECT> e <OPTION> 74
6.12. Grupos de opções: elemento <OPTGROUP> 75
6.13. Subgrupos de componentes <FIELDSET> e <LEGEND> 76
6.14. Botões HTML 4.0 <BUTTON> 76
6.15. Exercícios 77
7. Formulários e CGI 78
7.1. Programas CGI 78
7.2. Interligando formulários HTML e CGI 78
7.3. Tarefas básicas dos programas CGI 79
7.4. Variáveis de Ambiente 80
7.5. Exercícios 82
8. Princípios de HTTP 83
8.1. O que é HTTP 83
8.2. Métodos 83
8.3. Requisições HTTP e método GET 84
8.4. Respostas HTTP 85
8.5. Método HEAD 87
8.6. Execução de programas 88
8.7. Método POST 88
9. Uso de programas CGI 89
9.1. Tipos de programas CGI 89
9.2. Onde conseguir programas CGI 90
9.3. Como configurar e instalar os programas 90
9.4. Como criar programas CGI simples 91
9.5. Exercícios 95

6. Formulários

É possível construir, usando apenas HTML e nenhuma programação adicional, interfaces gráficas para entrada de dados através de uma página, exibida em um browser. Componentes como botões, chaves booleanas e campos para entrada de texto são criados usando o elemento HTML <INPUT> com atributos diferentes. Menus podem ser criados com o elemento <SELECT> que agrupa elementos <OPTION> contendo as suas opções. Existe ainda um elemento especialmente criado para a entrada de grandes quantidades de texto: o elemento <TEXTAREA>. Os componentes devem ser agrupados em blocos chamados de *formulários*, que contêm o endereço do programa, localizado em algum servidor Web, encarregado de fazer alguma coisa com os dados recebidos. Os blocos são delimitados pelo elemento <FORM>.

Embora sejam apenas três os elementos para entrada de dados (<INPUT>, <SELECT> e <TEXTAREA>), eles produzem 12 componentes gráficos diferentes. Existem outros seis elementos usados para agrupar componentes e criar botões, mas a maior parte deles não aparece no *Netscape* ou em browsers que não suportem HTML 4.0.

Qualquer elemento de entrada de dados só deve ser usado dentro de um bloco <FORM>. Embora um bloco <FORM> possa ter mais de um botão de envio, todos enviarão os dados para a única URL que ele indicar, no seu atributo ACTION.

As seções a seguir irão explorar cada um dos elementos usados para construir formulários no HTML 4.0. Se possível, tente repetir cada exemplo no seu editor de código, visualizando em seguida os efeitos no seu browser.

6.1. Elemento <FORM>

O elemento <FORM> é o mais importante dos elementos de formulário. Ele define o "bloco" de formulário que deve ser atrelado a um programa no servidor. A sua sintaxe, contendo os seus atributos (opcionais) mais importantes, está mostrada abaixo:

```
<FORM ACTION="URL para onde será enviado o formulário"
   METHOD="método HTTP (pode ser GET ou POST)"
   ENCTYPE="formato de codificação"
   TARGET="nome da janela que mostrará a resposta do formulário" >
        ... corpo do formulário (permite qualquer coisa permitida em <BODY>) ...
</FORM>
```

Embora todos os atributos do elemento FORM sejam opcionais, não é possível criar uma página que envie dados para o servidor sem que esse elemento contenha pelo menos o atributo: ACTION, que indica a URL do programa CGI (ou equivalente) que irá processar o conteúdo do formulário (o que o usuário digitar ou escolher).

Botões de envio

O bloco <FORM> vazio não mostra coisa alguma na tela. Para que ele possa pelo menos funcionar como um vínculo de hipertexto, ele precisa ter um botão de envio. Esse é o formulário mínimo: Um bloco <FORM> com um botão do tipo SUBMIT.

```
<FORM ACTION="/dados/tutorial.html">
   <INPUT TYPE="submit" VALUE="Tutorial de Formulários">
</FORM>
```

O botão SUBMIT, que é construído com o elemento <INPUT> não serve realmente para a entrada de dados como o nome do elemento sugere. É responsável apenas por executar a ação do formulário. O seu atributo TYPE é obrigatório para que tenha a aparência e funcionalidade do botão. O seu rótulo é definido pelo atributo VALUE. <FORM> e <SUBMIT> funcionam em conjunto. O trecho acima funciona de forma idêntica ao vínculo de hipertexto abaixo:

```
<A HREF="/dados/tutorial.html">Tutorial de Formulários</A>
```

porém o primeiro mostra um botão no lugar da âncora sublinhada que aparece no segundo. Ambos enviam a mesma requisição ao servidor:

```
GET /dados/tutorial.html HTTP/1.0
```

Mas formulários não foram criados para substituir vínculos de hipertexto. Eles servem principalmente para se enviar informações. O atributo METHOD indica o método de requisição HTTP que o browser utilizará para transferir as informações. Se ele estiver ausente, o método será GET, como vimos acima. Mas GET foi criado para o browser *receber* informação e não *enviá-la*. Não há bons motivos práticos para se usar outro método que não seja POST para enviar dados via formulários. Portanto, ACTION e METHOD são os atributos que devem estar sempre presentes em um bloco <FORM>, e METHOD deve conter o valor POST.

Entrada e envio de dados

Além do botão *Submit*, um formulário deverá conter outros componentes, já que o botão sozinho apenas executa o evento de enviar o formulário, mas não tem a capacidade de receber dados. Os outros elementos que são usados dentro do bloco <FORM>, e que podem armazenar informações que seguirão para o servidor, sempre têm um atributo obrigatório: NAME. Esse *nome* serve para identificar a *variável* que irá armazenar um *valor* fornecido pelo usuário/visitante da página Web. Depois de preencher um formulário e apertar um botão *Submit*, o nome da variável e o seu valor correspondente serão enviados ao servidor. Veja um exemplo:

```
<FORM ACTION="/cgi-bin/cadastro.pl" METHOD="GET">
    <P>Digite seu Nome: <INPUT TYPE="text" NAME="usuario">
       <INPUT TYPE="Submit" VALUE="Enviar para o servidor">
</FORM>
```

O método GET não é uma forma eficiente de se enviar dados, mas tem algumas vantagens didáticas, por isto está sendo usado neste exemplo. O elemento <INPUT>, acima, quando recebe o atributo TYPE com o valor "text" funciona como um campo para entrada de textos. Se não tiver outros atributos terá 20 caracteres de largura. Suponha agora que o usuário digite no campo de textos o nome "Florbela". Quando o usuário apertar o botão *Submit*, o browser construirá a seguinte requisição:

```
GET /cgi-bin/cadastro.pl?usuario=Florbela HTTP/1.0
```

O método GET envia os dados como parte da requisição, logo depois da URL do programa, separando-os por um sinal de interrogação "?". Isto é ineficiente e arriscado, pois, além deles ficarem visíveis, eles poderão ser truncados se as informações forem muito longas. Essa é a razão porque é melhor usar POST.

Mas, deixando de lado o método, observe como os dados do formulário foram montados pelo browser. Foi criado um par *nome=valor*. O *nome* é o identificador escolhido no atributo NAME. O *valor* é o texto que foi digitado.

Múltiplos campos de entrada de dados

Formulários geralmente servem para se enviar mais do que um nome. Atribuir um nome às variáveis passa a ter alguma utilidade quando há mais de um campo de entrada de dados:

```
<FORM ACTION="/cgi-bin/cadastro.pl" METHOD="GET">
    <P>Digite seu Nome: <INPUT TYPE="text" NAME="usuario">
    <P>Digite seu Número: <INPUT TYPE="text" NAME="numero">
    <INPUT TYPE="Submit" VALUE="Enviar para o servidor">
</FORM>
```

Este segundo bloco tem dois campos de entrada de dados. Suponha que o usuário preencha o formulário com as seguintes informações:

Digite seu Nome: Florbela

Digite seu Número: 47545

Desta vez a requisição do browser será a seguinte:

```
GET /cgi-bin/cadastro.pl?usuario=Florbela&numero=12345 HTTP/1.0
```

Observe que os dois pares nome/valor foram separados por um "&". Se houvesse mais campos de entrada de dados, os dados seriam agrupados da mesma forma, organizados em pares nome=valor e separados pelo "&". O arquivo cadastro.pl deve ser um programa executável e configurado para funcionar como CGI. Ele será responsável por separar os dados recebidos e fazer alguma coisa com eles.

Existem ainda outras conversões feitas pelo browser quando um formulário é enviado, como a conversão de caracteres especiais e espaços. Em uma URL, os caracteres :, /, ?, &, #, *, % e vários outros têm significado especial. URLs também não suportam caracteres acentuados nem espaços. Quando um usuário preenche um formulário contendo esses caracteres, é preciso que o browser os transforme antes de enviar. Para isto ele usa o código numérico do caractere dentro do alfabeto latino (alfabeto ISO-8859-1) em hexadecimal. Um "ç", por exemplo, é representado pelo número hexadecimal E7. O código do "ã" é E3 e assim por diante. O browser coloca um "%" antes do código, para indicar que o número não deve ser considerado literalmente, e monta a linha de dados da mesma forma como antes. Espaços são a única exceção. Em vez de serem convertidos no código %20, eles são substituídos pelo caractere "+". Suponha que os dados digitados no formulário sejam:

Digite seu Nome: Conceição da Silva

Digite seu Número: 333-01

Agora a requisição irá construir a seguinte linha de dados (que será enviada após a ? da URL do programa CGI se o método for GET):

```
usuario=Concei%E3%E7o+da+Silva&numero=333%2301
```

Mais uma vez, é o programa CGI que terá que "se virar" para decodificar os dados recebidos, localizando os sinais de percentagem, os sinais de + e fazendo a conversão.

Se o método usado em um formulário for POST, os dados serão codificados da mesma forma para envio ao servidor. A única diferença é a forma de envio. POST não envia dados através da URL da requisição. Os dados, que podem ser muito longos, são enviados em pedaços através do corpo da requisição do browser, e lidos pelo servidor na sua entrada padrão (porta de serviços). Um browser geraria algo parecido com a requisição a seguir:

```
POST /cgi-bin/cadastro.pl HTTP/1.0
Content-type: text/x-www-form-urlencoded
Content-length: 42

usuario=Concei%E3%E7o+da+Silva&numero=333%2301
```

Observe que os dados fazem parte do corpo da mensagem, que pode ter várias linhas se for necessário. O espaço reservado na URL da requisição é limitado a no máximo uns 2000 caracteres. Formulários mais longos, portanto, irão perder dados se enviados via GET. O tipo dos dados codificados e separados por "&" é text/x-www-form-urlencoded. É a forma mais simples. Essa codificação pode ser alterada com o atributo ENCTYPE de <FORM>.

Outros elementos de entrada de dados

Os elementos usados para entrada de dados via formulários são apenas <INPUT>, <SELECT> e <TEXTAREA>. Como vimos, <INPUT> também serve para construir botões. Mudando-se o atributo TYPE de <INPUT>, se obtém diferentes componentes de entrada de dados. São dez diferentes componentes.

O elemento <SELECT> representa uma caixa de seleção com uma ou mais opções contidas entre os descritores <OPTION> e </OPTION>. <TEXTAREA> representa uma caixa de texto. Todos os componentes de entrada de dados, sejam elementos <INPUT>, <SELECT> ou <TEXTAREA> devem ter um atributo NAME que contém o nome da variável que será usada pelo CGI para identificar as informações enviadas. Nas seções a seguir, cada componente será explorado em maiores detalhes.

6.2. Elemento <INPUT>

O elemento <INPUT> é usado para construir botões, e dispositivos de entrada de dados como caixas de texto e chaves booleanas (botões de rádio e caixas de checagem), componentes para o envio de arquivos e para armazenar variáveis invisíveis. O que distingue um objeto do outro é apenas o seu atributo TYPE. A sintaxe geral de <INPUT> é:

```
<input atributos>
```

onde os atributos podem ser diferentes, dependendo do valor do atributo TYPE, que pode ter um dos seguintes valores:

- type=**text** campo de entrada de texto
- type=**password** campo de entrada de texto oculto
- type=**radio** chave booleana para única seleção por grupo
- type=**checkbox** chave booleana para múltiplas seleções por grupo
- type=**submit** botão que implementa o evento "enviar dados"
- type=**reset** botão que implementa o evento "reinicializar formulário"
- type=**button** botão sem evento associado
- type=**image** imagem que implementa o evento "enviar dados"
- type=**hidden** campo oculto para armazenar pares nome/valor
- type=**file** componente para realizar o upload de arquivos

Cada um dos tipos de <INPUT> será ilustrado nas seções a seguir.

6.3. Botões *(TYPE=BUTTON, SUBMIT, RESET ou IMAGE)*

Os botões são criados com os atributos type=button, type=reset e type=submit. Usando o atributo type=image, o botão do tipo *Submit* poderá ser substituído por uma imagem. A sintaxe típica de um botão é:

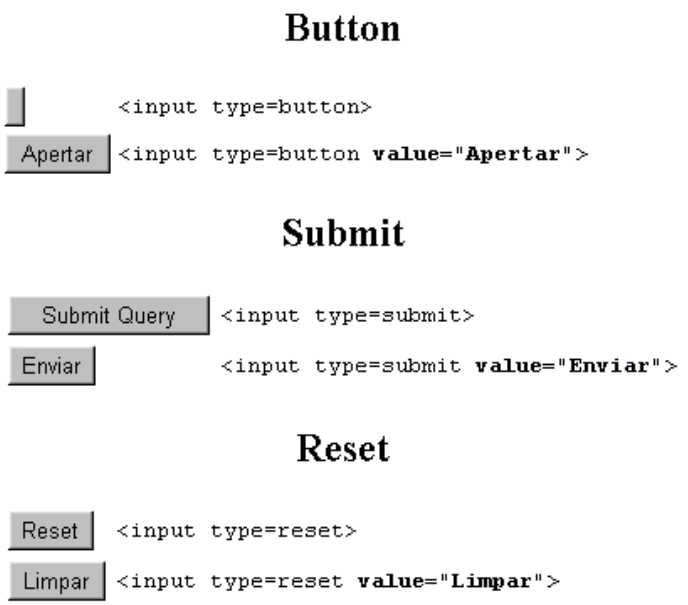
```
<INPUT TYPE="button" VALUE="rótulo do botão">    ou
<INPUT TYPE="submit" VALUE="rótulo do botão">    ou
<INPUT TYPE="reset" VALUE="rótulo do botão">
```

Se o atributo VALUE for omitido, o botão utilizará um rótulo padrão, escolhido pelo browser. A exceção são os botões do tipo *Button*, que não possuem um rótulo *default*. Para os botões do tipo *Submit* usando imagens, é necessário informar a URL onde reside a imagem através do atributo SRC:

```
<INPUT TYPE="image" SRC="http://a.com/imagens/botao.gif">
```

Os botões SUBMIT e RESET reagem a eventos. O botão BUTTON nada faz. Ele só tem utilidade se tiver um evento programado em JavaScript ou VBScript. Ao apertar o botão SUBMIT, o usuário envia os dados do formulário ao programa no servidor indicado pelo atributo ACTION de <FORM>. Ao apertar RESET, o usuário reinicializa o formulário, restaurando os valores iniciais de cada campo de entrada de dados.

O atributo VALUE permite alterar o texto que aparece dentro do botão. A figura ao lado mostra a aparência dos botões em um browser *Netscape Navigator* rodando em *Windows 95* com e sem o atributo VALUE.



O botão IMAGE envia como dados não só os valores digitados pelo usuário mas também as coordenadas em pixels da área da imagem que foi clicada. Isto permite que o programa CGI possa responder de forma diferente a partes diferentes da imagem.

A aparência dos botões, como sua borda, sua cor de fundo, a cor e o tamanho do texto, pode ser alterada por folhas de estilo. O suporte não é total no *Netscape*.

Há ainda vários novos atributos do HTML 4.0 que são apenas parcialmente suportados atualmente. O atributo ACCESSKEY aceita uma tecla que pode ser usada como atalho para o botão:

```
<INPUT TYPE=submit ACCESSKEY="s">
```

TITLE pode ser usado em botões para incluir informações adicionais de acessibilidade. Botões também podem ser desabilitados se tiverem o atributo DISABLED:

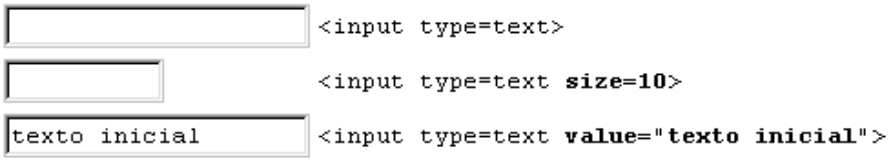
```
<INPUT TYPE=button VALUE="Não disponível" DISABLED>
```

6.4. Campos de texto de uma linha *(TYPE=TEXT ou PASSWORD)*

Campos de texto de uma única linha são definidos pelos componentes <INPUT> que possuem atributos TYPE com os valores PASSWORD ou TEXT. Ambos têm a mesma aparência, mas o texto dos objetos *Password* não é exibido na tela. No lugar dos caracteres digitados aparecem asteriscos (*Windows*) ou marcadores (*Macintosh*). A sintaxe de um elemento TEXT e seus principais atributos em HTML é a seguinte:

```
<INPUT TYPE="text"
       NAME="nome_do_campo_de_texto"
       VALUE="texto inicial do campo de textos"
       SIZE="número de caracteres visíveis"
       MAXLENGTH="número máximo de caracteres permitido">
```

Todos os atributos, exceto o atributo TYPE são opcionais. Se SIZE não for definido, a caixa de texto terá 20 caracteres de largura. Se MAXLENGTH não for definido, não haverá limite para o número de caracteres digitado no campo de textos. A figura abaixo ilustra a aparência de alguns componentes TEXT em um browser *Netscape 4.5* rodando em *Windows 95*.

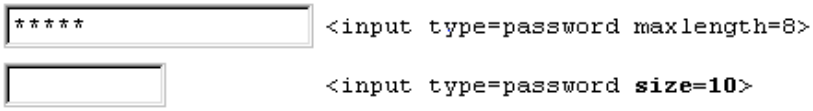


O atributo VALUE pode ser usado para definir o valor inicial do texto exibido. Esse valor poderá ser alterado pelo usuário.

O elemento do tipo PASSWORD é criado da mesma forma, mas com um atributo TYPE diferente:

```
<INPUT TYPE="password" ... >
```

Os caracteres do texto digitado em componentes PASSWORD não aparecem na tela, como mostrado na figura abaixo (*Windows95*):



Assim como os botões, campos de texto pode ter sua aparência alterada por folhas de estilo e conter os atributos ACCESSKEY, DISABLED, TITLE e outros que funcionarão em browsers compatíveis com HTML 4.0. Também foi introduzida no HTML 4.0 a possibilidade de tornar um campo de textos somente-leitura (para exibição de informações apenas). Isto é feito com o atributo READ-ONLY:

```
<INPUT TYPE=text VALUE="Mensagem inapagável" READ-ONLY>
```

6.5. Campos ocultos *(TYPE=HIDDEN)*

O componente do tipo HIDDEN é um campo de entrada de dados invisível, que o usuário da página não tem acesso. Serve para que o *autor da página* passe informações ao servidor, ocultando-as no código HTML da página. É muito utilizado para enviar informações de configuração para programas CGI usados por várias pessoas. Programas de envio de e-mail, por exemplo, podem obter o endereço destino (do autor da página) através de um

campo oculto. Esse campo também é muito usado na transferência de informações entre formulários que são formados por mais de uma página. Sua sintaxe é a seguinte:

```
<INPUT TYPE="hidden"
       NAME="nome_do_campo_oculto"
       VALUE="valor armazenado" >
```

Os atributos NAME e VALUE são obrigatórios para que o campo HIDDEN tenha alguma utilidade. A exceção é quando se usa programas como JavaScript, que rodam no próprio browser e têm condições de alterar o valor de tais campos durante a exibição da página.

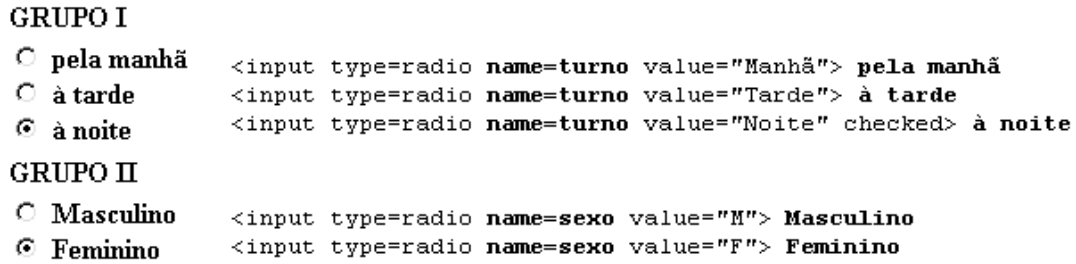
6.6. Chaves booleanas do tipo "botão de rádio" (TYPE=RADIO)

O componente do tipo RADIO representa um dispositivo de entrada booleano cuja informação relevante consiste em saber se uma opção foi selecionada ou não. Botões de radio são organizados em grupos de descritores com o mesmo nome (atributo NAME). Cada componente aparece na tela como um botão ou caixa de dois estados: *ligado* ou *desligado*. Dentro de um grupo de componentes (todos com o *mesmo* atributo NAME), somente um deles poderá estar ligado ao mesmo tempo. A sintaxe típica de um componente RADIO em HTML é a seguinte:

```
<INPUT TYPE="radio"
       VALUE="valor (o valor que será enviado ao servidor)"
       CHECKED     <!-- previamente marcado -->
       > Rótulo do componente
```

A figura abaixo mostra dois grupos de botões de rádio (em um browser *Netscape* rodando em *Windows95*). Observe que os atributos NAME distinguem um grupo do outro. O atributo CHECKED indica um botão previamente ligado mas que pode ser desligado pelo usuário ao clicar em outro botão.

Se não for definido um campo VALUE, com o valor a ser enviado ao servidor, será enviado o texto "on" ou "1". Se nenhum dos componentes de um grupo de botões de rádio for selecionado, a maior parte dos brow-



sers sequer envia o nome do elemento ao servidor, mas esse comportamento não é garantido (a especificação não prevê sua ocorrência), portanto, é uma boa idéia pré-marcar ou verificar, usando JavaScript, se um usuário marcou uma das opções antes que os dados sejam enviados ao servidor.

Todos os atributos HTML 4.0 aplicáveis aos botões podem ser usados nos botões de rádio. Usando folhas de estilo, poderá ser possível alterar a sua cor.

6.7. Elemento <LABEL>

Os botões de rádio não possuem um rótulo associado. O texto que o identifica deve ser colocado ao seu lado e não tem relação alguma com ele. O HTML 4.0 permite associar um rótulo de texto com elementos <SELECT>, <TEXTAREA> e <INPUT>, que inclui botões de rádio, usando o elemento <LABEL>:

```
<label accesskey="m">
   <input type=radio name=sexo value=M>Masculino
</label>
```

Em browsers que suportam esse recurso do HTML 4.0, será possível mudar o estado do botão de rádio clicando no texto ou digitando a letra "m" no teclado. Nos browsers que não suportam o recurso, o componente se comportará da mesma forma como sempre se comportou, ou seja, ignorando cliques do mouse sobre o rótulo do botão.

Nem sempre é possível ter o rótulo e o componente dentro do mesmo bloco <LABEL>. Uma situação é quando são usadas tabelas para organizar formulários. É comum o rótulo estar em uma célula e o componente em outra. Nessas situações não será possível ter os dois dentro do bloco <LABEL>. A solução nesses casos é usar <LABEL> com o atributo FOR. O componente precisará ser identificado através de sua propriedade ID e FOR informa ao browser que aquele rótulo pertence ao componente indicado. ID e NAME devem ter o mesmo valor, sempre:

```
<tr>
<td><label for="usuario">Nome: </label></td>
<td><input type=text name=usuario id=usuario></td>
</tr>
```

6.8. Chaves booleanas tipo "caixas de checagem" *(TYPE=CHECKBOX)*

Cada elemento <INPUT> do tipo CHECKBOX aparece na tela como um botão ou caixa que pode assumir dois estados: *ligado* ou *desligado*. Diferentemente dos componentes RADIO, vários componentes CHECKBOX de um mesmo grupo podem estar ligados ao mesmo tempo. Não há, portanto, sequer a necessidade de organizar tais objetos em um grupo.

A sintaxe de um elemento CHECKBOX em HTML é praticamente idêntica à de RADIO, mudando apenas o valor do atributo TYPE:

```
<INPUT TYPE="checkbox" ...
       VALUE="valor (o valor que será enviado ao servidor)"
       CHECKED     <!-- previamente marcado -->
       > Rótulo do componente
```

A figura abaixo mostra um grupo de caixas de checagem (em um browser *Netscape* rodando em *Windows*. O atributo CHECKED indica um botão previamente ligado mas que pode ser desligado pelo usuário ao clicar em outro botão.

☑ **Café** `<input type=checkbox name=refeicoes value="Café" checked> Café`
☐ **Almoço** `<input type=checkbox name=refeicoes value="Almoço"> Almoço`
☑ **Jantar** `<input type=checkbox name=refeicoes value="Jantar"> Jantar`

Assim como RADIO, o elemento CHECKBOX pode ser alterado por folhas de estilo e com atributos HTML 4.0 como ACCESSKEY e DISABLED. Pode também ser associado a um rótulo através do elemento <LABEL>.

Como dois elementos CHECKBOX podem ter o mesmo nome e ambos podem ter sido marcados pelo usuário, o programa CGI terá que separar os dois para evitar sobreposição. Os dois elementos marcados do desenho acima não são anulados pelo browser, mas são concatenados e enviados da forma:

```
refeicoes=Caf%E9&refeicoes=Jantar
```

Como os grupos não têm importância em elementos CHECKBOX, pode-se usar nomes diferentes em vez de ter que se preocupar com a separação das variáveis no CGI:

```
<input type=checkbox name=cafe>
<input type=checkbox name=almoco>
<input type=checkbox name=jantar>
```

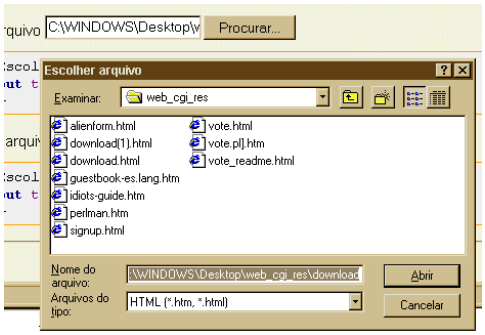
Os elementos marcados serão enviados como:

```
cafe=on&jantar=on
```

6.9. Upload de arquivos (TYPE=FILE)

O último elemento <INPUT> serve para realizar a transferência de arquivos da máquina do cliente até o servidor. Com o atributo TYPE=file, o elemento <INPUT> se transforma em um par de componentes que consiste de uma caixa de texto e um botão.

```
<label>Escolha um arquivo
    <input type=file name=arq2 title="Localizar arquivo">
</label>
```



O botão possui um rótulo que não é possível alterar. Sua ação, ao ser apertado, é abrir uma janela de escolha de arquivos através da qual o usuário poderá escolher o arquivo que deseja transferir. Uma vez escolhido o arquivo, o caminho do mesmo aparece na caixa de texto (veja figura).

Infelizmente, não é possível alterar a aparência ou o tamanho do elemento FILE de forma alguma. O campo de textos e o botão sempre têm tamanho fixo. Não pode haver texto inicial na caixa de texto nem outro rótulo para o botão.

A transferência de arquivos para o servidor exige que lá exista um programa CGI capaz de receber e tratar formulários no qual os dados são enviados em múltiplas partes. Não é possível enviar arquivos através da codificação padrão do formulário, que é do tipo text/x-www-form-urlencoded. Os dados têm que ser enviados usando a forma text/multipart-form-data, que deve ser expressa no atributo ENCTYPE. Esse formato, por necessitar de múltiplas linhas, não é suportado pelo método GET. Em suma, o bloco <FORM> de um formulário que contém elementos FILE deve ter a forma:

```
<FORM ACTION="/cgi-bin/programa.exe"
      METHOD=POST
      ENCTYPE="text/multipart-form-data"> ... </FORM>
```

6.10. Elemento <TEXTAREA>

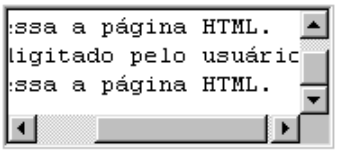
O elemento <TEXTAREA> é um bloco (possui descritor inicial e final) e define uma área onde se pode ler ou digitar texto em várias linhas. A sintaxe para criar um elemento <TEXTAREA> em HTML é a seguinte. Os atributos em negrito são obrigatórios e definem a altura e largura da área de texto:

```
<TEXTAREA
   ROWS="número de linhas visíveis"
```

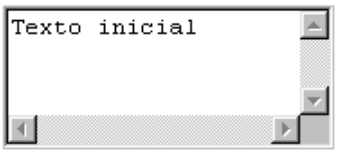
```
    COLS="número de colunas visíveis"
    NAME="nome_do_campo_de_texto"
    WRAP="hard | soft | virtual | physical | off">
         Texto inicial
</TEXTAREA>
```

Não é possível controlar a aparência exata dos elementos de texto como o <TEXTAREA> e <INPUT TYPE=TEXT> porque eles são dimensionados não em pixels mas em número de caracteres e linhas visíveis. As fontes usadas mudam de um browser para outro fazendo, por exemplo, que um campo <TEXTAREA> apareça 30 a 40% maior em browsers *Netscape* que no *Internet Explorer*. Pode-se usar folhas de estilo para controlar a aparência mas elas só funcionarão no *Internet Explorer*.

A figura abaixo mostra a aparência de componentes <TEXTAREA> no *Netscape*:




```
<textarea rows=3 cols=20></textarea>
```




```
<textarea rows=3 cols=20>Texto inicial</textarea>
```

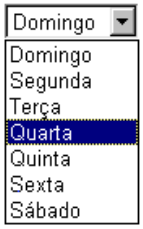
O atributo WRAP pode ser usado para controlar a maneira como o texto é enviado e visualizado no campo de texto. Com WRAP=off, o texto rola lateralmente e as linhas não quebram. Com WRAP=soft ou WRAP=virtual as linhas quebram na tela mas chegam no servidor como uma única linha. WRAP=hard ou WRAP=physical introduz os caracteres de novas-linhas que farão com que as quebras de linha cheguem ao servidor.

Elementos <TEXTAREA> podem ser desabilitados com o atributo DISABLED e tornados somente-leitura com o atributo READ-ONLY. Esses recursos, assim como alguns recursos de folhas de estilo, só são suportados por browsers que oferecem suporte amplo ao HTML 4.0.

6.11. Elementos <SELECT> e <OPTION>

Menus e listas de seleção como as mostradas nas figuras ao lado são criadas com o elemento HTML <SELECT>. Este componente não pode conter texto. Contém apenas elementos <OPTION> que representam as opções do menu.

Os componentes <SELECT> podem ter uma aparência e comportamento diferente dependendo se possuem ou não os atributos SIZE e MULTIPLE. A figura ao lado ilustra o efeito desses atributos, transformando um menu simples em uma lista com barra de rolagem lateral que permite a seleção de múltiplas opções. SIZE altera o número de linhas visíveis. MULTIPLE, se presente, habilita a seleção de




```
<select name=umdia>
    <option value="D">Domingo</option>
    <option value="S">Segunda</option>
    <option value="T">Terça</option>
    <option value="Q">Quarta</option>
    <option value="I">Quinta</option>
    <option value="X">Sexta</option>
    <option value="B">Sábado</option>
</select>
```

```
<select name=variosdias size=4 multiple>
    <option value="D">Domingo</option>
    <option value="S">Segunda</option>
    <option value="T">Terça</option>
    <option value="Q">Quarta</option>
    <option value="I">Quinta</option>
    <option value="X">Sexta</option>
    <option value="B">Sábado</option>
</select>
```

mais de uma opção. A sintaxe de um elemento HTML <SELECT> com seus principais atributos está mostrada abaixo:

```
<SELECT NAME="nome_do_componente"
   SIZE="número de opções visíveis"
   MULTIPLE        <!-- Suporta seleção múltipla -->
 >
      <OPTION ...> Opção 1 </OPTION>
      ...
      <OPTION ...> Opção n </OPTION>
</SELECT>
```

Todos os atributos são opcionais. A existência do atributo NAME é obrigatória em formulários que terão dados enviados ao servidor. Cada um dos elementos <OPTION> poderá ter a seguinte sintaxe básica:

```
<OPTION VALUE="Valor da opção" SELECTED >
      Texto descrevendo a opção
</OPTION>
```

O atributo VALUE informa o que deve ser enviado ao servidor quando a opção correspondente tiver sido selecionada. Mas VALUE é opcional. Na ausência de um elemento VALUE o browser envia o texto visível, contido entre <OPTION> e </OPTION>. Esse texto não será enviado se VALUE estiver presente.

O atributo SELECTED marca a opção como previamente selecionada. Nos casos onde só é permitida a seleção simples, apenas uma das opções deverá conter o atributo.

Nos casos onde se permite seleção múltipla, os pares nome/valor são repetidos e o comportamento é semelhante ao caso de objetos CHECKBOX de mesmo nome. Por exemplo, se na lista de opções mostrada o usuário tiver selecionado os dias terça e quarta, o browser enviará:

```
variosdias=T&variosdias=Q
```

Folhas de estilo podem ser aplicadas tanto no objeto SELECT como um todo, como nas suas opções individuais, onde se pode mudar a cor de fundo, a cor e tamanho da fonte usada, etc. Um elemento <SELECT> também pode ser desligado usando DISABLED. Todos esses recursos só são suportados por browsers que implementam HTML 4.0.

6.12. Grupos de opções: elemento <OPTGROUP>

Nos browsers que implementam HTML 4.0, é possível organizar grupos de opções <OPTION> através do elemento <OPTGROUP>, também aceito dentro de um bloco <SELECT>. Esse elemento cria um rótulo e endenta ou destaca de alguma outra forma as opções de menu que contém, criando submenus.

```
<select name=cores>
  <option value="nenhuma">Escolha uma cor</option>
  <optgroup title=luminosas>
    <option value="R">Vermelho</option>
    <option value="G">Verde</option>
    <option value="B">Azul</option>
    <option value="W">Branco</option>
  </optgroup>
  <optgroup title=opacas>
    <option value="C">Azul Ciano</option>
```

```
    <option value="M">Magenta</option>
    <option value="Y">Amarelo</option>
    <option value="K">Preto</option>
  </optgroup>
</select>
```

6.13. Subgrupos de componentes <FIELDSET> e <LEGEND>

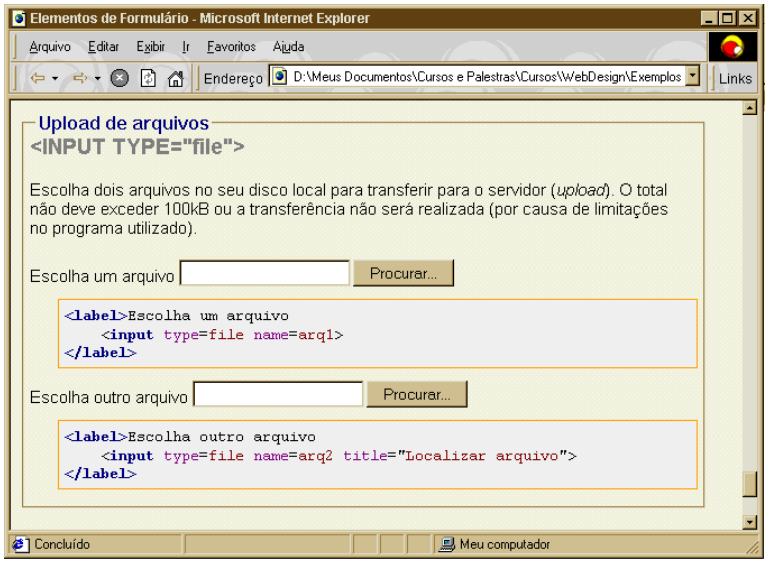
Em formulários longos, pode ser necessário identificar suas partes com informações específicas para indicar a organização de um grupo de elementos. O HTML 4.0 oferece os elementos `<FIELDSET>` e `<LEGEND>` para essa finalidade.

`<FIELDSET>` deve ser colocado em volta dos elementos que se quer agrupar. `<LEGEND>` é usada para identificar o subgrupo.

```
<fieldset> <!-- HTML 4.0! Pode ñ funcionar em alguns browsers! -->
   <legend>Upload de Arquivos</legend>

   <!--  vários elementos de formulário  -->

</fieldset>
```



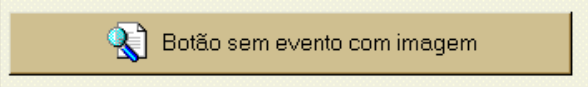
6.14. Botões HTML 4.0 <BUTTON>

Você deve ter achado estranho um botão que não serve para a entrada de dados ser construídos a partir de um elemento chamado `<INPUT>`. O grupo de trabalho que desenvolve o HTML também e por isso propuseram um novo elemento que tem a finalidade apenas ser um botão e que oferece diversas vantagens sobre os botões tradicionais.

O elemento `<BUTTON>` pode ter seu rótulo definido através do texto que contém ou através do atributo `VALUE`. Possui um atributo `TYPE` que pode assumir os valores `submit`, `reset` e `button`.

Com <BUTTON>, fica simples criar um botão que contém uma imagem. Basta colocar a imagem dentro do bloco <BUTTON>:

```
<button title="Botão compatível com HTML 4.0" type="button">
    <img src="localizar.gif" width="33" height="30" align=absmiddle>
    Botão sem evento com imagem
</button>
```



É preciso ter cuidado já que o botão é um elemento que pode fazer falta se não aparece na página. Browsers que não suportam HTML 4.0 poderão não mostrar nada no lugar onde deveria estar o botão, portanto, vale a pena esperar por um suporte mais amplo antes de deixar de usar os antigos <INPUT TYPE=SUBMIT>, <INPUT TYPE=RESET> e <INPUT TYPE=BUTTON>.

6.15. Exercícios

1. Pratique com os componentes de formulário mostrados neste capítulo. Faça uma ou mais páginas HTML que utilizem todos os elementos. Veja o resultado em browsers diferentes.
2. Construa um formulário de feedback para sua página. O formulário deverá ter um lugar para que o visitante deixe seu nome, seu e-mail e mande uma mensagem. Você pode acrescentar outros campos se quiser.
3. Faça o formulário de feedback acima funcionar. Se você tem conta no IBPINET, informe "/cgi-bin/FormMail.pl" como sendo o ACTION do seu formulário e inclua um campo HIDDEN com o nome "recipient" e valor contendo o seu e-mail. Agora qualquer visitante poderá mandar e-mail para você via Web.
4. Mude a página de resposta do exercício anterior acrescentando outro campo hidden, desta vez com o nome "redirect". O campo VALUE deverá conter o nome da página que será mostrada após o envio do formulário.

7. Formulários e CGI

Para nada serve um formulário se não há um programa no servidor capaz de fazer alguma coisa com os dados preenchidos pelo cliente. Dependendo do servidor, de sua plataforma e configurações, o programa pode ser uma página ASP, PHP ou JSP, pode ser um servlet Java, um arquivo *.dll, um arquivo *.exe, um programa em Perl, um roteiro Shell ou outro mecanismo que seja capaz de interagir com o formulário.

A forma mais universal de interação com programas iniciados pelo servidor é o mecanismo chamado de CGI, suportado por quase todos os servidores Web. Este capítulo mostra o que é preciso fazer para que um programa CGI obtenha informações de um formulário.

7.1. Programas CGI

CGI é um serviço oferecido pelo servidor. É uma extensão ao serviço HTTP. Servidores que suportam CGI podem ser configurados a executarem programas externos ao software servidor, passar-lhes parâmetros recebidos de uma requisição do browser e redirecionar sua saída como uma resposta HTTP.

A finalidade do CGI é permitir que o servidor funcione como ponte (ou *gateway*) para aplicações externas. A especificação CGI define regras para a construção de aplicações que poderão ser executadas pelo servidor HTTP. As regras só se aplicam à entrada e saída das aplicações. É uma especificação de "caixa-preta". Não interessa o conteúdo. As aplicações podem ser escritas em qualquer linguagem desde que sejam capazes de gerar corretamente o final do cabeçalho do servidor (imprimir uma linha em branco).

Um programa CGI, para ser útil, precisa fazer algo mais. O cabeçalho de resposta é deixado incompleto pelo servidor porque ele espera que o programa CGI gere dados de algum tipo. Como o servidor não tem como saber que tipo de dados o CGI irá gerar, é o programa CGI que terá que dar essa informação, construindo (imprimindo) uma linha de cabeçalho "Content-type" antes de encerrar o cabeçalho com uma linha em branco.

Para tratar formulários, o programa tem que ser capaz de ler eficientemente os dados na entrada, identificando o método usado pelo browser, extraindo as informações da requisição do browser (se o método tiver sido GET) ou da entrada padrão (se tiver sido POST) e depois decodificando as informações geradas pelo browser.

Por não exigir o conhecimento de uma linguagem de programação específica, CGI tornou-se uma solução extremamente popular, mesmo depois que surgiram tecnologias mais eficientes e mais simples para programar a Web. Existem diversas aplicações comerciais ou gratuitas na Internet que foram escritas em C, C++, Perl, AppleScript, Java, Visual Basic, Delphi, Bourne-shell e várias outras linguagens rodando em todas as plataformas onde há servidores Web.

7.2. Interligando formulários HTML e CGI

Como vimos em um capítulo anterior, a interface de formulários oferecida por HTML permite que o leitor de uma página Web possa enviar informações de volta para o servidor através de componentes como botões, caixas de checagem, caixas de texto, menus, etc.

Cada componente de entrada de dados de um formulário tem um atributo `NAME` e pode ter um atributo `VALUE`. `NAME`, geralmente um nome escolhido pelo autor da página, contém o `nome` que será utilizado pelo programa CGI para recuperar o `valor` que o leitor da página enviou. Esse valor também pode ser fixado pelo autor da página através dos atributos `VALUE` de cada componente. Como vimos, cada um dos elementos de um formulário envia seus dados no formato `nome=valor`.

Ao enviar um formulário com mais de um campo de entrada de dados, todos os pares nome/valor são concatenados com um "`&`". Os espaços também são convertidos em "+" e caracteres reservados convertidos em hexadecimal, da forma `%hh`. Essa é a forma padrão de codificação dos dados de um formulário feita pelo browser e representada pelo tipo MIME `text/x-www-form-urlencoded`.

Tome como exemplo o formulário mostrado na figura abaixo. Quando o leitor apertar o botão, o browser enviará a seguinte requisição ao servidor:

```
POST /cgi-bin/bd.exe HTTP/1.0
Content-type: text/x-www-form-urlencoded
Content-length: 36

nome=Jo%E3o+da+Silva&idade=23&cor=yellow
```

```
<FORM ACTION="/cgi-bin/bd.exe"
      METHOD="POST">
<p>Nome: <input type=text name=nome>
<p>Idade: <input type=text name=idade>
<p>Escolha uma cor:
<select name=cor>
  <option value=red>Vermelho</option>
  <option value=yellow>Amarelo</option>
  <option value=blue>Azul</option>
</select>
<p><input type=submit value="Enviar">
<input type=reset value="Limpar">
</FORM>
```

Esses dados são enviados ao servidor que os repassa ao programa localizado em `/cgi-bin/bd.exe`. A requisição também poderia ter sido realizada usando o método `GET`, mas nesse caso os dados seriam enviados na linha de requisição após a "?" em uma parte da URL chamada de *Query String*:

```
GET /cgi-bin/bd.exe?nome=Jo%E3o+da+Silva&idade=23&cor=yellow HTTP/1.0
```

7.3. Tarefas básicas dos programas CGI

Programas que seguem a especificação CGI e irão ler dados enviados por formulários devem ser capazes de obter e decodificar uma linha como

```
nome=Jo%E3o+da+Silva&idade=23&cor=yellow
```

A obtenção dos dados geralmente também requer a capacidade de ler a entrada padrão, possivelmente ler um arquivo ou conexão de rede e obter informações armazenadas em variáveis de ambiente definidas pelo servidor.

A decodificação consiste em:

- converter os códigos hexadecimais `%hh` nos seus caracteres equivalentes

- converter todos os "+" em espaços
- separar os pares nome/valor pelo "&"
- identificar, para cada par nome/valor, o nome (à esquerda do "=") e o valor (à direita) e montar uma associação que permita obter o valor a partir do nome.

Uma vez extraídos os dados, o programa CGI pode realizar qualquer tarefa no servidor. Pode, por exemplo, iniciar uma outra aplicação, montar uma requisição em linguagem SQL para envio a um banco de dados ou recuperar uma imagem de um dispositivo externo. No final, o programa deve retornar para o cliente uma imagem, uma página ou outra seqüência de bytes qualquer. O servidor não tem como saber o tipo da seqüência de bytes para montar o cabeçalho para o cliente, então, deixa-o incompleto para que seja completado pelo programa CGI. O programa precisa, portanto, imprimir na sua saída no pelo menos:

- uma linha de cabeçalho "`Content-type: tipo_MIME`", informando o tipo de dados enviados de volta ao servidor (por exemplo, `Content-type: text/html`)
- uma linha em branco, indicando o fim do cabeçalho
- os dados (página HTML, código GIF, código JPEG, código de máquina, etc)

Supondo que o programa CGI seja escrito em linguagem C, para imprimir uma mensagem em HTML com o texto "Hello CGI World" em resposta a uma requisição de um browser, ele teria que conter as linhas:

```
printf("Content-type: text/html\n\n");   /* \n\n quebra linha duas vezes */
printf("<html><body><h1>Hello CGI World</h1></body></html>");
```

7.4. Variáveis de Ambiente

Um programa CGI precisa freqüentemente obter várias informações sobre o servidor, sobre o sistema operacional local e sobre o browser e máquina cliente que fez a requisição. Toda requisição do browser faz com que o servidor defina algumas variáveis de ambiente, que geralmente duram o mesmo tempo da transação.

Essas variáveis são como variáveis de ambiente do DOS ou do Unix. Se você escrever um programa CGI como um roteiro Shell, por exemplo, poderá imprimir o nome do browser que fez o pedido com o seguinte programa:

```
#!/bin/sh

echo "Content-type: text/html"
echo ""
echo "<h1>Teste de CGI</h1>"
echo "<p>O seu browser é " $HTTP_USER_AGENT
```

A variável `HTTP_USER_AGENT` é criada pelo servidor durante a transação. Como qualquer outra variável Unix, ela pode ser lida usando o cifrão antes do nome (se você tivesse criado um programa em MS-DOS, a variável seria lida como `%HTTP_USER_AGENT%`).

Variáveis de ambiente são definidas tanto por servidores Unix como por servidores Windows. Nem todas são suportadas em todos os servidores. As que são suportadas pela maioria dos servidores estão listadas abaixo:

SERVER_SOFTWARE
O nome e a versão do software do servidor que responde ao pedido. *Formato*: nome/versão. *Exemplo*: Apache/1.2

SERVER_NAME
O nome da máquina onde roda o servidor (DNS) ou o endereço IP. *Exemplo*: www.servidor.org

GATEWAY_INTERFACE

A número da revisão da especificação CGI que o servidor utiliza. *Formato*: CGI/revisão

CONTENT_LENGTH

O comprimento do conteúdo enviado pelo cliente.

CONTENT_TYPE

Tipo do conteúdo dos dados para buscas que têm informação anexada.

DOCUMENT_NAME

Nome do documento (se programa for um documento HTML).

DOCUMENT_ROOT

Caminho absoluta da raiz de documentos HTML.

DOCUMENT_URI

URL (URI) do documento ou do programa CGI.

DATE_LOCAL

Data e hora locais. *Exemplo*: Friday, Aug 13, 1999 0:0:0 GMT-03

DATE_GMT

Data e hora locais no formato *Greenwich Mean Time*. *Exemplo*: Thursday, Aug 12, 1999 21:0:0 GMT

LAST_MODIFIED

Data da última modificação do documento ou do programa.

HTTP_COOKIE

Lista de cookies para o caminho e domínio atuais, separados por ponto-e-vírgula.

HTTP_ACCEPT

Tipos MIME que o cliente aceitará, dados por cabeçalhos HTTP. Cada item desta lista deve ser separado por vírgulas. *Formato:* type/subtype, type/subtype, etc.

HTTP_USER_AGENT

O browser que o cliente está usando para enviar o pedido.

HTTP_REFERER

URL da página que contém o link para a página atual.

PATH

Caminho (de subdiretórios).

PATH_INFO

Informação extra de caminho (de subdiretórios), como fornecida pelo cliente.

PATH_TRANSLATED

Versão de PATH_INFO traduzida pelo servidor.

QUERY_STRING

A informação que segue a ? na URL que referencia o programa CGI. É a informação de busca (query).

QUERY_STRING_UNESCAPED

A mesma informação contida em QUERY_STRING, mas com os caracteres de escape (%nn) traduzidos.

REMOTE_HOST

O nome da máquina que faz o pedido. Se o servidor não tem esta informação, não deve estabelecer um valor para esta variável, mas para REMOTE_ADDR.

REMOTE_ADDR

O endereço IP da máquina remota que faz o pedido.

REQUEST_METHOD

O método com o qual o pedido foi feito. Para HTTP, esse método é "GET", "HEAD", "POST", etc.

SERVER_PROTOCOL

O nome e a revisão do protocolo de informações utilizado pelo pedido. *Formato*: protocolo/revisão. *Exemplo*: HTTP/1.0

SERVER_PORT
O número da porta para o qual foi enviado o pedido.

SERVER_ROOT
O diretório absoluto da localização do servidor na rede.

SCRIPT_NAME
Um caminho virtual para o programa CGI que está sendo usado.

7.5. Exercícios

1. Escreva um programa CGI usando uma linguagem de sua escolha (no Unix, você pode escolher C, shell ou Perl) que imprima as variáveis de ambiente definidas pelo browser. Não precisa gerar HTML. Pode gerar text/plain (o programa só funcionará se a primeira linha imprimir "Content-type: text/plain" e a segunda for em branco). Transfira o programa para o diretório CGI do seu servidor Web e teste-o. (A forma de imprimir uma variável de ambiente em Perl é

   ```
   print $ENV{'NOME_DA_VARIÁVEL'};
   ```

   Não se esqueça de imprimir o cabeçalho *antes*.

2. Rode o formulário-exemplo localizado no site do curso (http://site_do_curso/allform.html), preencha-o e veja o resultado.

8. Princípios de HTTP

Este é um capítulo de referência. Não é preciso conhecer os assuntos tratados aqui para desenvolver ou usar CGI.

8.1. O que é HTTP

HTTP significa *Hypertext Transfer Protocol*. É o principal protocolo utilizado nas comunicações na Internet e o protocolo que dá sustentação ao serviço conhecido como *World Wide Web*. A especificação, elaborada pelo W3C[1], o define como:

> *HTTP é um protocolo de nível de aplicação para sistemas de hipermídia, colaborativos e distribuídos. É um protocolo genérico, sem estado e orientado a objetos que pode ser usado para diversas tarefas, tais como servidores de nomes e sistemas de gerenciamento de objetos distribuídos, através da extensão de seus métodos de requisição [RFC2068]*

A W3C define HTTP como um protocolo sem estado (*stateless*). Isto quer dizer que o protocolo não preserva informações de estado entre requisições ao servidor. Se um servidor recebe uma seqüência de 3 requisições de três clientes ao mesmo tempo ele não saberá separar as requisições por cliente de forma a considerar informações da primeira requisição que podem influenciar na segunda ou terceira. Isto quer dizer que não será possível, por exemplo, implementar aplicações que dependam de informações de estado definidas em páginas separadas, como aplicações de comércio online, sem recorrer a mecanismos externos ao protocolo.

8.2. Métodos

HTTP define um conjunto de mensagens chamados de *métodos de requisição HTTP* que são usados pelo cliente, geralmente um browser como o *Netscape Navigator* ou *Microsoft Internet Explorer*, para enviar uma requisição ao servidor. Esta requisição é, na maior parte das vezes, uma solicitação para que o servidor devolva uma página HTML, uma imagem, um componente ou recurso multimídia. O método HTTP usado neste tipo requisição é o método **`GET`** que, além de requisitar páginas, imagens ou programas, pode requisitar informações geradas na saída de um dispositivo ou programa executado pelo servidor.

Vários outros métodos são definidos pelo protocolo HTTP. O protocolo também garante a possibilidade de extensões. A versão HTTP 1.1 define 7 métodos básicos, que são: `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `TRACE`, `OPTIONS`. Um servidor Web mínimo, que suporte HTTP 1.1, deve ser capaz de entender pelo menos os métodos `GET` e `HEAD`. Um servidor Web típico recebe normalmente requisições `GET`, `HEAD` e `POST`, sendo a grande maioria requisições `GET`.

As mensagens passadas em HTTP podem ser *requisições* ou *respostas*. As requisições são formadas e enviadas por um cliente HTTP. As respostas são retornadas por um servidor. A porta de comunicações onde o servidor aguarda requisições é, por *default*, a porta 80. Se outra porta for utilizada, ela deverá constar da requisição.

[1] World Wide Web Consortium (http://www.w3.org)

8.3. Requisições HTTP e método GET

Uma requisição HTTP obedece a um formato padrão que consiste de uma linha de requisição, linhas de cabeçalho, uma linha em branco e talvez, linhas de dados:

```
Método   URL_solicitada   Protocolo/versão
Propriedade: valor
Propriedade: valor ...

Várias linhas de dados (informações enviadas pelo browser)
```

A primeira linha é a mais importante. Ela usa um método HTTP para operar sobre uma determinada URL, e diz qual o protocolo e versão que deve ser usado para a resposta.

O cabeçalho pode conter várias linhas com informações adicionais essenciais ou não. As linhas sempre são da forma Propriedade: valor e o formato de todo o cabeçalho é padronizado pela especificação Internet RFC822, de define o formato para cabeçalhos de e-mail. Essa especificação declara que o cabeçalho termina quando o receptor encontra uma linha em branco. Tudo o que segue será considerado informação a ser consumida (dados).

Tipicamente, as linhas de cabeçalho contêm o nome da máquina destino (se separado da URL na requisição), a porta de serviços, a data, o nome e versão do cliente, etc. A maior parte das requisições HTTP não envia dados adicionais após o cabeçalho. As que enviam dados precisam ainda informar o tamanho e tipo dos dados enviados para que o servidor saiba o que fazer com eles.

O método `GET` tem a finalidade de recuperar informações do servidor. É o método mais usado pelo browser. A especificação [RFC2068] o define como um método seguro e *idempotente*, ou seja, deve produzir o mesmo resultado se repetido várias vezes. `GET` nunca envia dados após o cabeçalho e possui a seguinte sintaxe mínima:

```
GET URL_relativa HTTP/1.1
Host: nome_da_maquina
                        <--- linha em branco (CRLF)
```

Por exemplo:

```
GET /docs/index.html HTTP/1.1
Host: www.festa.org.br
```

A requisição acima faz uma conexão à porta 80 da máquina `www.festa.com.br` e solicita ao servidor o arquivo `/docs/index.html`. A requisição acima normalmente contém várias outras linhas de cabeçalho antes da linha em branco. Abaixo está uma provável requisição para buscar a página `index.html`, formada ao se clicar em um vínculo de hipertexto no *Microsoft Internet Explorer*:

```
GET /docs/index.html HTTP/1.1
User-Agent: Mozilla 4.0 (compatible; MSIE 4.01; Windows98)
Accept: image/gif, image/x-bitmap, image/jpeg, image/pjpeg, *.*
Referer: http://www.casa.com.br/pagina2.html
Host: www.festa.com.br
```

As propriedades de cabeçalho presentes dependem do método usado na requisição. Algumas das propriedades freqüentes em requisições GET são:

- **`User-Agent`**: Contém uma linha que identifica o browser que faz o pedido.
- **`Host:`** Endereço ou nome do servidor.

- **`Date:`** Data da requisição no formato GMT ou local.
- **`Accept:`** Pode haver vários cabeçalhos Accept. Cada um contém um tipo MIME (tipo/subtipo) que é aceito pelo browser que faz o pedido.
- **`Cookie`**: Envia os cookies relacionados com o domínio e caminho atual.
- **`Accept-Encoding:`** Tipos de codificação aceitas pelo browser.
- **`Referer:`** Contém uma linha com a URL do documento onde está a referência (link) do recurso solicitado. Não aparece se a requisição não foi causada por um link.
- **`If-Modified-Since:`** Usada com o método GET para torná-lo condicional. Se o documento não mudou desde a última vez em que foi recuperado, ele não será enviado e o browser deve buscá-lo no seu diretório de arquivos Internet temporários.

8.4. Respostas HTTP

Após receber uma requisição, um servidor HTTP sempre responde. A resposta pode consistir de uma mensagem de erro, de redirecionamento ou de sucesso. A sintaxe das respostas é:

```
Protocolo/Versão   Código   Informações de status
Propriedade: valor
Propriedade: valor ...

Várias linhas de dados (informações enviadas pelo servidor)
```

A resposta começa com uma linha que informa o estado da resposta. O código de *status* é um número de três dígitos que pode ter a forma `1xx`, `2xx`, `3xx`, `4xx` ou `5xx`, onde `x` é qualquer dígito de `0` a `9`. Códigos começando em `4` ou `5` indicam mensagens de erro. Iniciando em `2` indicam sucesso. `3xx` é usado para indicar requisição incompleta ou redirecionamento. `1xx` é usado para retornar informações durante o processamento.

Os códigos de erro `4xx` indicam que o *cliente* deve ter errado (o cliente poderá corrigir o erro, se souber como). Os códigos `5xx` indicam que o *servidor* está consciente que o erro é dele, e não pode ser solucionado pelo cliente (o servidor inclui possíveis programas CGI e arquivos .htaccess). As informações de *status* contêm um texto descrevendo a ocorrência. Veja o significado de alguns códigos:

- **`OK 200`** – A requisição foi atendida.
- **`No Response 204`** – O servidor recebeu a requisição mas não há nada a ser retornado. Browser deve se manter na mesma página.
- **`Bad request 400`** – A sintaxe do pedido era inadequada ou impossível de ser satisfeita.
- **`Unauthorized 401`** – O cliente não tem autorização para obter a informação
- **`Forbidden 403`** – O objeto não pode ser enviado. Não adianta ter autorização.
- **`Not found 404`** – O servidor não encontrou nada que combinasse com a URL que recebeu.
- **`Internal Error 500`** – Erro interno (pode ser erro em programas CGI, erro em arquivos .htaccess, etc.)
- **`Not implemented 501`** – Este recurso não foi implementado no servidor
- **`Moved 301`** – A informação se mudou para outro lugar (houver redirecionamento). A resposta contém a nova URL onde o recurso pode ser localizado.
- **`Not Modified 304 –`** A página não foi modificada desde o último acesso. Recupere-a do diretório de arquivos da Internet temporários.

Por exemplo, uma resposta iniciando em:

```
HTTP 1.0 404 Not Found
```

será enviada ao browser caso o servidor não consiga atender à requisição do browser por não encontrar o recurso solicitado, ou

```
HTTP 1.0 200 OK
```

em caso de sucesso.

Como resposta à requisição `GET` acima (seção anterior), o servidor poderia ter retornado:

```
HTTP/1.0 200 OK
Server: Netscape-FastTrack/2.0a
Date: Mon, 4 Jan 1999 18:37:47 GMT
Content-type:text/html
Content-length: 19920
Last-modified: Sun, 13 Dec 1998 5:11:59 GMT

<!DOCTYPE ... </html> (19920 bytes de informação)
```

Poderia também ter retornado um erro. Por exemplo, ao requisitar uma página HTML ao `servle-trunner` (servidor especial da Sun que só serve servlets e não aceita pedido de páginas) ele retorna um erro:

```
HTTP/1.0 403 Forbidden
Server: servletrunner/2.0
Content-type:text/html
Content-length: 135
Date: Mon, 4 Jan 1999 18:39:55 GMT

<html><head><title></title></head>
<h1>403 Forbidden</h1><body>
Will not serve files, only servlets
</body></html>
```

Observe a linha em branco separando os dados (o conteúdo de uma página) do cabeçalho (gerado pelo servidor). Isto ocorre em qualquer resposta do servidor.

Os cabeçalhos que o servidor devolve ao cliente referem-se a propriedades do objeto retornado, mais informações sobre o ambiente. A maioria deles são gerados automaticamente pelo servidor. Para criar uma aplicação CGI, é preciso completar o cabeçalho com uma linha em branco, pois nessas condições, o servidor não o completa. Antes de completar o bloco do cabeçalho, porém, o programador CGI pode redefinir ou acrescentar novos cabeçalhos, como o cabeçalho "Content-type", que informa o tipo do conteúdo gerado pelo CGI. Alguns dos principais cabeçalhos de resposta estão listados a seguir:

- **`Content-Length:`** Conteúdo em bytes da informação enviada.
- **`Content-Type:`** Indica o tipo de mídia dos dados que estão sendo enviados. Isto pode ser o tipo MIME ou tipos de múltiplas partes como: Multipart/alternative, Multipart/related, Multipart/mixed ou Multipart/parallel *Exemplo*: Content-type: text/html.
- **`Content-Encoding:`** Codificação utilizada (se utilizada). Formatos suportados pelo padrão W3C são x-compress, x-gzip, x-www-form-urlencoded.
- **`Date:`** Indica a data de criação do objeto devolvido. O formato deve seguir a especificação RFC850 (datas GMT).
- **`Server:`** Inclui informações sobre o servidor como o seu nome e versão.

- **`Expires:`** Indica uma data em que a informação no documento não é mais válida (usada por browser que fazem *cache*). O formato é o mesmo de Date. Um usuário poderá fazer com que o CGI imprima um cabeçalho Expires com a data atual ou no passado para evitar que ele seja armazenado em pastas de arquivos temporários (*cache*).
- **`Last-Modified:`** Este cabeçalho indica a data e a hora da última modificação de um recurso. O formato é igual do de Date. O browser usa esta informação para guardar os arquivos no *cache*.
- **`Location:`** Define a localização exata de um recurso, através de uma URL. Este deve ser usado em separado de qualquer outro. Se o browser ler uma linha `contendo Location: http://novaURL.com/novapag.html`, a janela vai ser redirecionada para buscar as informações na nova URL.
- **`Set-Cookie:`** Define (cria) um cookie. *Por exemplo*: `Set-Cookie: usuario=fulano`. Esta propriedade poderá ser usada várias vezes, para criar mais de um cookie.
- **`Pragma:`** Pode ter o valor "no-cache" indicando que o servidor deve retornar um documento mesmo que ele não tenha mudado (usado pelo *Reload* do browser).

Entre as propriedades da cabeçalho do servidor, a propriedade `Content-type` é uma das mais importantes. Ela informa ao browser o tipo dos dados que ele está recebendo. Sem essa informação, o browser não seria capaz de distinguir uma página HTML de uma imagem GIF. O cabeçalho `Content-type` contém o tipo MIME dos dados que virão a seguir. O cabeçalho `Content-length`, não menos importante, informa quantos caracteres o browser terá que ler antes que toda a informação seja recebida.

Sempre que se cria uma página dinâmica, produzida por um programa CGI, é necessário montar (imprimir) a parte do cabeçalho HTTP que o servidor não faz sozinho (o Content-Type, por exemplo) e terminar o bloco com uma linha em branco.

O servidor Web sempre retorna alguma coisa. Qualquer coisa que esteja armazenada nos diretórios de documentos do servidor HTTP será retornada para o browser se houver permissão. Pode ser uma página HTML, mas também pode ser um arquivo executável *Windows*. Se a raiz de documentos (/) de um servidor Web rodando em *Windows* for `C:\WINDOWS\COMMAND` e o browser solicitar

```
GET /format.com HTTP/1.0
```

o servidor devolverá o arquivo para download (jamais o executará):

```
HTTP/1.0 200 OK
Server: Caranguejeira/1.0
Date: Mon, 4 Jan 1999 21:10:33 GMT
Content-type: application/x-msdownload
Content-length: 41604
Last-modified: Sat, 24 Aug 1996 11:11:00 GMT

é ¨Converted____MZx_U_I_ ___ÿÿ__ð (...)
```

8.5. Método HEAD

O método `HEAD` é semelhante ao método `GET`. A única diferença entre os dois é a resposta do servidor. O servidor que recebe `HEAD` retorna o cabeçalho completo dos dados requeridos mas não retorna os dados.

`HEAD` é normalmente usada para verificar a data de última modificação do arquivo, o tamanho do arquivo, tipo dos dados, etc. para fins de otimização da transferência ou gerenciamento de *cache* ou *proxy*.

8.6. Execução de programas

A maior parte das requisições de um browser geralmente são para recuperar algum recurso no servidor. Às vezes, porém, o cliente quer que o servidor execute um programa. É preciso que o servidor esteja *configurado* para identificar o que o cliente quer a partir da URL da requisição. Um servidor que suporte CGI, por exemplo, pode identificar um pedido de execução se a URL solicita um recurso contido em um diretório especial:

```
GET /cgi-bin/format.com HTTP/1.0
```

neste exemplo chamado de `/cgi-bin` (é preciso que o servidor esteja configurado para reconhecer `/cgi-bin` como um diretório que *não* fornece documentos, mas inicia a execução de programas.) Desta vez, supondo o mesmo servidor *Windows* do exemplo anterior, o programa será executado na máquina servidora e a saída do programa poderá ser retornada para o cliente (essa segunda parte provavelmente não ocorrerá nesse caso pois o arquivo não gera os cabeçalhos CGI.).

A extensão do nome de um arquivo também é freqüentemente usada pelo servidor para distinguir um recurso estático de um recurso executável. Em servidores Microsoft IIS:

```
GET /texto.asp HTTP/1.0
```

fará com que o servidor tente interpretar comandos especiais na página HTML `texto.asp` e envie o resultado para o browser porque ele foi instruído a analisar o código de arquivos terminados em `.asp` e tentar interpretá-los antes de devolvê-los para o browser.

Geralmente um programa no servidor requer parâmetros enviados pelo cliente para poder executar corretamente. Tome por exemplo um contador que precisa saber o nome do arquivo onde armazenar a contagem. A linha de informações que segue a "?" em uma URL tem um comprimento limitado e é chamada de *Query String* e pode ser usada para passar parâmetros. A requisição ao contador poderia ser da forma:

```
GET /cgi-shl/count.pl?arq=pg34.txt HTTP/1.0
```

O programa em Perl `count.pl` será então executado pelo servidor recebendo o parâmetro `arq=pg34.txt` como argumento.

8.7. Método POST

`POST` é usado pelo cliente para enviar dados ao servidor durante uma requisição. O alvo da requisição deve ser um agente capaz de realizar alguma coisa com os dados recebidos. A URL da requisição `POST`, portanto, geralmente contém informações que fazem o servidor executar algum procedimento em vez de simplesmente retornar uma imagem ou página.

`POST` é indicado para enviar grandes quantidades de dados ao servidor já que não é limitado como o *Query String* usado para enviar informações via `GET`. O método `POST` não precisa ser seguro ou *idempotente*. Pode provocar alterações nos dados e duas requisições não precisam retornar os mesmos dados.

Veja um exemplo com o contador mostrado anteriormente. Se o browser enviasse uma requisição `POST` em vez de `GET` os dados seriam enviados da seguinte maneira:

```
POST /cgi-shl/count.pl HTTP/1.0
Content-type: text/plain
Content-length: 12

arq=pg34.txt
```

Mas o browser não usa `POST` a não ser que seja instruído a fazê-lo. A forma mais comum de fazer isto, é usar formulários HTML.

9. Uso de programas CGI

Para rodar programas CGI é preciso configurar o servidor e, muitas vezes, os próprios programas. Este capítulo distingue vários tipos de programas CGI e mostra como um Web designer pode usá-los sem precisar saber nada sobre a linguagem de programação na qual foram escritos.

A maior parte dos provedores de acesso possui diversos CGIs mais populares já instalados e disponíveis através de um diretório cgi-bin central. Antes de transferir e tentar instalar um programa, procure saber se o seu provedor já não possui o serviço. Programas que são geralmente disponibilizados pelos provedores incluem contadores, leitores de formulários, programas que enviam e-mail, livros de visitas e sistemas de busca local. Se o seu provedor já possui um desses programas instalados, descubra como usá-los. Pode ser tão simples como escrever a URL do programa no campo ACTION do seu formulário.

Se você decidir aprender CGI para criar aplicações, terá que aprender uma linguagem de programação. Nas seções finais deste capítulo, alguns programas CGI escritos em linguagens interpretadas são analisados. Nessas seções será mostrado como alterar programas e como criar aplicações simples sabendo-se um mínimo de programação.

9.1. Tipos de programas CGI

Programas CGI podem ser interpretados ou compilados. Geralmente não é possível descobrir qual a linguagem em que foram escritos os programas compilados. Eles são os mais fáceis de instalar. Alguns (os comerciais) possuem uma rotina de instalação (tipo *InstallShield*). Outros só precisam ser copiados para um diretório executável para começarem a funcionar. Grande parte dos programas necessita de configuração, mesmo que seja através do formulário HTML.

Programas interpretados são mais fáceis de modificar, já que o código-fonte está sempre disponível. A desvantagem é que eles são mais difíceis de configurar. Não basta copiar o programa para um diretório executável. É preciso associá-lo ao interpretador e, muitas vezes, realizar configurações adicionais no servidor Web.

Programas em MS-DOS e Shell

Pode-se criar programas CGI simples usando a linguagem MS-DOS, interpretada pelo programa Command.COM do Windows. MS-DOS é uma linguagem muito limitada e pode-se tornar complexa e ineficiente quando usada para gerar programas longos. Ela é útil apenas para testes em servidores que rodam em Windows.

Alguns servidores poderão ser incapazes de rodar programas em MS-DOS já que tais aplicações rodam de maneira diferente das aplicações Windows. Programas MS-DOS só podem ser instalados em servidores que distinguem CGI do Windows de CGI do MS-DOS.

Shell é a linguagem de linha de comando do Unix. Apesar da semelhança inicial, é bem mais poderosa que MS-DOS e permite a criação de programas bem mais sofisticados. Mas um programa em Shell pode ficar muito complexo e ineficiente se for muito grande e realizar muitas tarefas repetitivas. Grandes programas em Shell costumam precisar usar recursos de outras linguagens como AWK e Tcl/Tk no Unix.

Shel é, porém, útil para realizar testes em servidores e para rotinas curtas e simples. Antes da adoção de Perl como linguagem preferencial, a maior parte dos programas CGI disponíveis na Web eram escritas em Shell.

Programas em Perl

Perl é a linguagem mais popular usada no desenvolvimento de programas CGI interpretados. Ela possui vários recursos que auxiliam o desenvolvimento de tais aplicações e é interpretada, facilitando a modificação e a sua portabilidade. Existem interpretadores Perl para os sistemas Windows, Unix, Macintosh e diversos outros.

Diversos sites na Internet disponibilizam programas CGI em Perl que, geralmente, podem ser usados gratuitamente. Muitos são configuráveis sem que seja preciso mexer no código. Além de ser aplicado em CGI, Perl também é bastante usado na administração de sistemas Unix e Windows (substituindo o editor de registro). A última versão estável é Perl 5.005. A implementação do Perl no Windows pode ser obtida na ActiveState (www.activestate.com).

Para instalar programas CGI em Perl é preciso associar o endereço do interpretador com o programa fonte. A forma de realizar essa associação varia entre servidores e sistemas operacionais.

9.2. Onde conseguir programas CGI

Há vários sites na Internet que disponibilizam programas CGI gratuitamente para download. Outros cobram taxas após um período de uso. Os melhores programas permitem a instalação e configuração sem que seja preciso alterar uma linha sequer do código. Costumam ser distribuídos com um manual (em HTML) e exemplos de aplicações completas (páginas HTML integradas aos programas). A documentação informa ao usuário como configurar o programa para que possa rodá-lo no seu servidor.

Abaixo está uma lista de alguns sites que distribuem programas CGI. Na página dedicada a este curso você encontrará outros endereços e programas prontos como contadores, leitores de formulários, livros de visitas, sistemas de busca, entre outros.

- www.freescripts.com/ - diversos scripts CGI de domínio público
- www.geocities.com/SiliconValley/Orchard/6104/ - CGIs gratuitos e comerciais.
- cgi-lib.berkeley.edu/ - biblioteca para construção de aplicações CGI em C e Perl
- www.cpan.org/ - repositório de módulos Perl
- www.cgi-resources.com/ - diversos scripts em Perl, C, C++, Tcl, AppleScript
- perfect.com/freescripts/ - diversos programas gratuitos
- www.nethernet.com/free/ - diversos programas gratuitos
- www.cgi-world.com/ - repositório de programas CGI em várias linguagens
- www.worldwidemart.com/scripts/ - programas úteis em C e em Perl

9.3. Como configurar e instalar os programas

Instalar programas compilados é simples: basta copiar o programa para um diretório que permita a sua execução pelo sistema operacional. Programas interpretados são mais complicados de instalar pois requerem a existência entre o código-fonte e o interpretador. Não há uma forma padrão de realizar essa associação. Vários servidores a fazem de maneiras diferentes. No Unix, que faz a associação é sempre o sistema operacional. No Windows, na maior parte das vezes é preciso criar uma associação no Windows e no servidor. No Personal Web Server a associação precisa ser realizada no Registro do Windows.

Para programas escritos em Perl para execução através do servidor Apache, o caminho até o programa interpretador deverá ser feita na primeira linha, após os caracteres #!. Quem acha o interpretador para o programa no Apache Linux é o sistema operacional. Já no Windows, é o próprio Apache quem o localiza.

Para instalar um programa CGI em Perl escrito por outra pessoa, é necessário descobrir qual o endereço onde está instalado o seu interpretador Perl. A forma mais fácil de fazer isto é perguntar ao seu administrador do sistema. Locais típicos no Linux são /usr/local/bin/ e /usr/bin/. Se o programa estiver localizado no primeiro subdiretório, a primeira linha do programa, portanto, deverá ser alterada para:

```
#!/usr/local/bin/perl
```

O Apache Windows associa o programa ao interpretador da mesma forma. Suponha que o Perl esteja instalado em c:\perl\. O programa executável deverá estar em c:\perl\bin\, portanto, a primeira linha do programa deverá conter:

```
#!c:\perl\bin\perl.exe
```

Há outras variáveis que talvez precisem ser mudadas. Um bom script de propósito geral contém documentação suficiente para explicar cada configuração necessária. Geralmente tais alterações necessárias são realçadas com comentários e aparecem bem no início do programa. A maioria requer que o usuário digite informações sobre o sistema que está utilizando, como endereço do interpretador, caminho completo de seu diretório CGI, nome e IP de sua máquina, endereço do programa de e-mail, localização do banco de dados, etc.

Programas de propósito geral também costumam requerer configuração especial na página HTML, geralmente através de elementos <input type=hidden>. Os melhores programas são totalmente configuráveis através de tais campos e por isso podem ser reutilizados mais de uma vez e por autores diferentes.

As seções a seguir mostrarão como configurar um dos mais populares programas para tratamento de formulários e seu envio por e-mail. O roteiro FormMail.pl disponível gratuitamente em *Matt's Script Archive* (www.worldwidemart.com/scripts/).

Configuração e uso do FormMail.pl

Consulte anexos distribuídos no site deste curso.

Alterações no código-fonte

Os programas distribuídos pela Internet geralmente são bastante configuráveis pelo autor de um site através de HTML pois destinam-se a um público que não pode ou não quer mexer com programação. Mesmo assim, há vários programas configuráveis que não são flexíveis o suficiente. Por exemplo, o `FormMail`, embora possa sempre ser redirecionado para páginas escolhidas pelo autor da página que contém o formulário, ainda mostra mensagens em inglês que não podem ser alteradas a menos que se manipule o código. A própria mensagem enviada para quem preencheu o formulário contém partes em inglês que não se pode remover sem ter acesso ao código.

Não basta mudar o texto contido nos comandos `print` para alterar as mensagens. Perl coloca variáveis (palavras começando com `$`) dentro do texto e removê-las acidentalmente poderá causar erros. Essas alterações no código fogem ao escopo deste curso, mas, se você decidir explorar a linguagem Perl (apêndice desta apostila), não deixe de voltar e analisar o código desse programa. Ele se tornará bem mais simples.

9.4. Como criar programas CGI simples

É fácil criar programas CGI simples com um conhecimento mínimo de linguagens de programação. Ir além disso requer uma dedicação maior aos recursos oferecidos pela linguagem. Nesta seção serão apresentados pequenos programas que listam variáveis de ambiente e isolam variáveis passadas via formulário. Eles requerem conhecimentos mínimos de Bourne Shell (linha de comando Unix), Perl (veja apêndice desta apostila) e Visual Basic (não abordada neste curso). Não são analisados exemplos de programas que salvam em disco nem conectam-se a bancos de dados pois tais recursos não dependem de CGI mas da linguagem utilizada.

Listagem das variáveis de ambiente em Shell e MS-DOS

O programa a seguir lista diversas variáveis de ambiente geradas pela requisição do browser. É um programa escrito em *Bourne Shell*, a linguagem usada para interpretar a linha de comando do Unix. Observe que na primeira linha foi informado o caminho até o interpretador. Isto é essencial. Também é necessário mudar o bit executável (usando chmod) para que o programa rode de forma independente.

O programa abaixo não gera HTML, por isso o tipo de dados é text/plain. Se você quiser mudar o tipo para text/html não se esqueça de imprimir também a marcação HTML como elementos <P> antes de cada parágrafo e um elemento <h1> em volta do título.

```
#!/bin/sh

echo "Content-type: text/plain"
echo ""
echo "Variáveis de ambiente"
echo "----------------------"

echo "SERVER_SOFTWARE: " $SERVER_SOFTWARE
echo "SERVER_NAME: " $SERVER_NAME
echo "GATEWAY_INTERFACE: " $GATEWAY_INTERFACE
echo "CONTENT_LENGTH: " $CONTENT_LENGTH
echo "CONTENT_TYPE: " $CONTENT_TYPE
echo "DOCUMENT_NAME: " $DOCUMENT_NAME
echo "DOCUMENT_ROOT: " $DOCUMENT_ROOT
echo "DOCUMENT_URI: " $DOCUMENT_URI
echo "DATE_LOCAL: " $DATE_LOCAL
echo "DATE_GMT: " $DATE_GMT
echo "LAST_MODIFIED: " $LAST_MODIFIED
echo "HTTP_COOKIE: " $HTTP_COOKIE
echo "HTTP_ACCEPT: " $HTTP_ACCEPT
echo "HTTP_USER_AGENT: " $HTTP_USER_AGENT
echo "HTTP_REFERER: " $HTTP_REFERER
echo "PATH: " $PATH
echo "PATH_INFO: " $PATH_INFO
echo "PATH_TRANSLATED: " $PATH_TRANSLATED
echo "QUERY_STRING: " $QUERY_STRING
echo "QUERY_STRING_UNESCAPED: " $QUERY_STRING_UNESCAPED
echo "REMOTE_HOST: " $REMOTE_HOST
echo "REMOTE_ADDR: " $REMOTE_ADDR
echo "REQUEST_METHOD: " $REQUEST_METHOD
echo "SERVER_PROTOCOL: " $SERVER_PROTOCOL
echo "SERVER_PORT: " $SERVER_PORT
echo "SERVER_ROOT: " $SERVER_ROOT
echo "SCRIPT_NAME: " $SCRIPT_NAME
echo "------------------------------"
```

Para rodar o programa, depois que ele estiver instalado em um diretório CGI, simplesmente chame-o através de uma URL no seu browser. Utilize-o para receber dados de formulários e veja os diferentes resultados.

O programa acima poderá rodar também em sistemas *Windows* se for salvo em um arquivo de texto com a extensão `.BAT`. Para isto é necessário que as variáveis de ambiente sejam chamadas não precedidas por um `$` mas entre `%` e `%`. A primeira linha também deve ser eliminada pois ela não faz sentido em MS-DOS. Não é preciso identificar o local do interpretador pois o interpretador `Command.COM`, que interpreta o MS-DOS é chamado automaticamente quando a extensão do arquivo é `*.BAT`. Deve-se substituir a primeira linha por:

```
@echo off
```

Para que não seja impressa na página enviada para o browser o nome dos comandos (`echo`) e o *prompt* do MS-DOS. Também é necessário substituir as linhas

```
echo ""
```

que imprime uma linha em branco no Unix, por

```
echo.
```

que faz o mesmo em MS-DOS.

Alguns servidores Web têm problemas ao rodar arquivos `.BAT`. Eles não rodam em servidores que suportam apenas Win-CGI. É preciso ter um diretório `cgi-bin` que esteja preparado para rodar CGI via MS-DOS. Uma melhor alternativa para o desenvolvimento de programas CGI em *Windows* é a linguagem *Visual Basic* ou *Perl*.

Listagem de variáveis com Visual Basic

Programas em Visual Basic devem ser compilados antes que sejam utilizados em CGI, usando a biblioteca `CGI.BAS` (ou outra equivalente). O código abaixo lista os dados enviados através de um formulário HTML (os pares nome/valor).

```
Sub CGI_main()
    Dim loop as Integer
    Dim info as String

    Send ("Content-type: text/html")
    Send ("")
    Send ("<h1>Dados digitados no formulário</h1>")

    For loop = 0 to CGI_NumFormTuples - 1
        Send ("<p><b>" & CGI_FormTuples(loop).key)
        Send ("</b>: " & CGI_FormTuples(loop).value)
    Next loop
End Sub
```

O programa compilado pode ser armazenado em qualquer diretório que suporte Win-CGI como o diretório `cgi-bin` do *Personal Web Server*. As funções e variáveis iniciadas em `CGI_` (em negrito) fazem parte da biblioteca `CGI.BAS`.

Perl

No capítulo anterior havia um pequeno formulário HTML que enviava para o servidor as variáveis nome, idade e cor. O seu código está repetido abaixo:

```
<FORM ACTION="/cgi-bin/bd.pl" METHOD="POST">
<p>Nome: <input type=text name=nome>
```

```
<p>Idade: <input type=text name=idade>
<p>Escolha uma cor:
<select name=cor>
  <option value=red>Vermelho</option>
  <option value=yellow>Amarelo</option>
  <option value=blue>Azul</option>
</select>
<p><input type=submit value="Enviar">
<input type=reset value="Limpar">
</FORM>
```

Em Perl há uma biblioteca de funções para CGI que permite a decodificação automática dos dados. É a biblioteca `cgi-lib.pl`[2]. O programa a seguir, escrito em Perl, usa essa biblioteca para transformar os dados recebidos e colocá-los em uma variável Perl que chamamos de `%dados`. Essa variável é capaz de armazenar pares nome valor que podem ser recuperados através da sintaxe `$dados{'nome'}`. Isto quer dizer que, para armazenar o valor contido no atributo `NAME` "cor" do HTML em uma variável chamada `$cor`, pode-se fazer

```
$cor = $dados{'cor'};
```

O programa `bd.pl` listado abaixo faz isto. Primeiro importa a biblioteca (com a instrução `require`). Depois chama a função `ReadParse`, que obtém os dados da entrada padrão e coloca os pares nome/valor no vetor `%dados`. Em seguida obtém o valor de cada atributo (nome, idade e cor). No final, imprime o cabeçalho `Content-type` e gera uma página HTML contendo os dados digitados.

```
#!/usr/local/bin/perl

require "cgi-lib.pl";
&ReadParse(\%dados);

$nome = $dados{'nome'};
$idade = $dados{'idade'};
$cor= $dados{'cor'};

print "Content-type: text/html\n\n";

print "<h1>Dados enviados</h1> \n";
print "<p>Nome: $nome\n";
print "<p>Idade: $idade\n";
print "<p>Cor: $cor\n";
```

As informações devolvidas em uma página HTML também poderiam ser salvas em disco ou enviadas por e-mail.

Para rodar o programa acima no *Apache Windows*, basta mudar a primeira linha informando o endereço do interpretador Perl instalado. Em outros servidores isto pode não ser necessário.

---

[2] Per 5 possui um módulo chamado CGI:: que faz a mesma coisa e é tão simples de usar quanto o cgi-lib.pl. Embora o módulo CGI:: seja parte da distribuição do Perl 5, decidimos não usá-lo porque ele não está disponível em algumas distribuições mais antigas do Perl para Windows e Macintosh.

9.5. Exercícios

1. Instale a página de feedback que você fez em exercício de um capítulo anterior no seu servidor Web local (Apache). Escreva um programa CGI em Perl (como o do exemplo acima) que leia os dados digitados e os imprima na tela.
2. Faça um construtor de frases: um formulário HTML que receba várias palavras e escolhas do usuário e construa uma frase com ela, devolvendo uma página HTML com a frase construída.

Criação de Web Sites II

3

SSI
Cookies
Programação

Conteúdo

10. Server-side includes *104*
10.1. Como habilitar SSI no servidor 104
10.2. SSI em servidores Apache 105
Instalação do suporte a SSI 105
Controle da segurança 105
10.3. SSI em servidores Microsoft 106
10.4. Instruções mais comuns 106
10.5. Exemplos 107
Página de testes 107
Data de modificação 108
Repositórios de imagens ou outros arquivos 109
Cabeçalhos e rodapés 109
10.6. Exercícios 110
11. Cookies *111*
11.1. Criação de cookies via cabeçalhos HTTP 111
11.2. Alteração do tempo de vida 112
11.3. Espaço de nomes de um Cookie 112
11.4. Parâmetros 113
11.5. Criação de cookies via HTML 113
11.6. Recuperação de cookies 114
11.7. Programas CGI que criam cookies 114
11.8. Programas CGI que recuperam cookies 115
11.9. Exercícios 116
12. Princípios de programação *117*
12.1. Tipos de linguagens 117
Linguagens procedurais 117
Linguagens declarativas 118
Linguagens baseadas ou orientadas a objetos 118
12.2. Componentes de um programa 118
Algoritmo: fluxo de controle 118
Variáveis 119
Subrotinas 119
Exercício 121
Estruturas de dados e objetos 121
Exercício 122
Operadores e expressões 122
Exercícios 123
Estruturas de controle de fluxo 123
Exercícios 124
12.3. Como escrever um programa 124
O código-fonte 124
O programa-objeto 124
A execução 125

10. Server-side includes

SERVER-SIDE INCLUDES (SSI) são instruções para o servidor embutidas em uma página HTML. Normalmente, um servidor não analisa o conteúdo de uma página HTML, mas simplesmente a recupera e envia para o cliente após uma requisição. Se o suporte a SSI estiver instalado e a página for identificada como uma página que o servidor deve analisar (*server-parsed HTML*), ele irá tratar a página como se fosse um programa. Tudo o que for HTML puro será copiado para a saída padrão (como já era feito antes) mas se o servidor encontrar instruções especiais dirigidas a ele, da forma:

```
<!--#include file="texto_extra.txt" -->
```

ele irá tentar executá-las. Elas parecem comentários HTML, mas jamais devem chegar ao browser. O servidor deverá substitui-las pelo resultado da sua execução. As instruções sempre causam a inclusão de *texto* que o servidor copiará para a saída padrão. No final, o browser deverá receber uma página cujo conteúdo foi alterado dinamicamente, e cujo código difere daquele contido no arquivo HTML armazenado no disco do servidor.

Caso, por algum motivo, o servidor não abra o arquivo para interpretar os comandos SSI, eles chegarão ao browser mas não serão exibidos na página, pois estarão entre comentários HTML.

10.1. Como habilitar SSI no servidor

As páginas que podem sofrer transformações precisam ser identificadas pelo servidor. Normalmente, quando não está configurado o suporte a SSI, o servidor simplesmente copia o conteúdo de um arquivo solicitado na saída padrão, e não interpreta seu código.

Como o código só pode ser incluído em páginas HTML (e não em imagens, programas, etc.), deve-se restringir a procura pelos comandos SSI apenas a esses arquivos, identificados pela extensão do nome do arquivo. O servidor então tentará processar todos os comandos encontrados nessas páginas antes de enviar a nova página gerada dinamicamente ao browser. O desempenho do servidor, porém, deverá cair, pois o ele terá que analisar todas as páginas devolvidas à procura de comandos SSI, mesmo que elas não tenham tais comandos.

Uma solução para limitar a análise e interpretação apenas aos arquivos que realmente contêm instruções SSI seria identificá-los. Uma forma de identificação poderá ser através de localização, como fizemos para que o servidor identificasse programas CGI. Cria-se um diretório e determina-se que qualquer página localizada naquele diretório será analisada em busca de instruções SSI. Mas arquivos que contém SSI são geralmente documentos, que fazem parte de coleções de documentos. Separá-los das outras páginas seria inconveniente.

Uma outra solução, que é a mais usada, é identificar os arquivos que contém instruções SSI através de uma extensão de nome de arquivo diferente. Pode ser qualquer uma, mas convencionalmente utiliza-se a extensão .shtml.

O suporte a SSI varia entre servidores Web. Praticamente todos suportam as instruções <!--#include -->. Outros suportam instruções proprietárias e poderão suportar um pequeno grupo de instruções populares nos servidores Unix. A configuração de SSI em alguns servidores é desnecessária. Basta mudar a extensão do arquivo para .shtml que ele passará a abrir as páginas para ler os comandos. Em outros servidores é preciso ativar o servi-

ço, pois alguns comandos poderão oferecer riscos de segurança. Na seção seguinte mostraremos como configurar o suporte a SSI nos servidores mais populares.

10.2. SSI em servidores Apache

Há duas configurações necessárias no Apache para que ele passe a suportar SSI. Uma deve ser realizada pelo administrador do servidor, que tem acesso aos arquivos *.conf. A outra, poderá ser realizada em arquivos .htaccess.

Instalação do suporte a SSI

É preciso configurar o servidor para que o ele trate de forma diferente arquivos .shtml (que ele os analise e não devolva simplesmente). É preciso também dizer ao servidor que o arquivo .shtml deverá ser devolvido ao browser como text/html. Se isto não for feito, o browser poderá receber o código-fonte da página.

Para dizer ao servidor que ele deve analisar um certo arquivo, é preciso defini-lo como "server-parsed" (analisado pelo servidor). Essa alteração é realizada no arquivo srm.conf (ou httpd.conf). O trecho abaixo diz ao servidor que arquivos com extensão `.shtml` devem ser analisados:

```
AddHandler server-parsed .shtml
```

É preciso agora acrescentar, ao tipo de dados text/html, a extensão de arquivo .shtml, para que o servidor possa gerar corretamente o Content-type no cabeçalho de resposta:

```
AddType text/html .shtml
```

Ainda falta configurar a segurança. Normalmente, SSI está desligado em servidores Apache. É preciso permitir a interpretação dos comandos.

Controle da segurança

Esta parte pode ser configurada em dois lugares: no `access.conf` (ou `httpd.conf`), acessível pelo administrador do servidor, ou no `.htaccess`, sob o controle do administrador do site. A alteração é realizada na diretiva `Options` para cada diretório afetado (se for aplicado na raiz de documentos afetará todo o site.

Os servidores *Apache* permitem dois níveis de segurança para SSI. O mais baixo permite a execução de todos os comandos suportados pelo servidor. O outro suporta todos os comandos com exceção do comando `<!--#EXEC -->` que inclui texto na página mediante a execução de um programa no servidor. A seguir, um trecho do `access.conf` (ou `httpd.conf`) correspondente ao diretório raiz de documentos (`DocumentRoot`), permitindo a execução de todos os SSI inclusive o `<!--#EXEC -->`.

```
<Directory /usr/apache/htdocs>
   Options Indexes FollowSymLinks Includes ExecCGI
   AllowOverride None
   order allow,deny
   allow from all
</Directory>
```

Para não permitir a execução de `<!--#EXEC -->` basta omitir a opção `ExecCGI`. Se `Includes` não estiver presente, nenhum tipo de SSI funcionará.

Para configurar o suporte a SSI no .htaccess, basta acrescentar as opções na linha Options. Um arquivo `.htaccess` mínimo que permite qualquer tipo de SSI no seu diretório contém:

```
Options Includes ExecCGI
```

Para que páginas SSI funcionem, basta agora terem a extensão .shtml e conterem comandos SSI.

10.3. SSI em servidores Microsoft

Esta edição provisória não mostra como configurar SSI em servidores Microsoft. Aguarde a publicação de um anexo no site do curso.

10.4. Instruções mais comuns

Os comandos SSI geralmente são dependentes do fabricante do servidor, ou dos *plug-ins* suportados pelo servidor. Há, porém, uma coleção de instruções disponíveis na maioria dos servidores modernos. Teste os comandos abaixo no seu servidor. Não se esqueça de verificar antes se o serviço está habilitado e se a página possui a extensão de nome de arquivo correta.

<!--#ECHO *atributo*-->

Inclui o conteúdo de uma variável de ambiente CGI no documento.

Atributos:

- `VAR="variável-de-ambiente"` armazena o nome da variável de ambiente a retornar. *Exemplo:*

  ```
  <!--#ECHO VAR="LAST_MODIFIED"-->
  ```

  inclui na página a data da última modificação do documento. Há várias variáveis de ambiente que só são visíveis via SSI.

<!--#INCLUDE *atributo*-->

Adiciona todo o conteúdo de um arquivo ao documento, antes de enviá-lo ao cliente. Suporta um atributo, que pode ser `FILE` ou `VIRTUAL`. Este é o comendo SSI de maior suporte. Funciona na maioria dos servidores Windows e Unix.

Atributos:

- `FILE="arquivo"`. Informa a localização de arquivo no sistema de arquivos do sistema operacional local (caminho relativo ou absoluto).

  ```
  <!--#INCLUDE FILE="c:\paginas\htdocs\docs\extra.txt" -->
  ```

- `VIRTUAL="arquivo"`. Informa a localização de arquivo no sistema de arquivos do servidor (caminho relativo ou absoluto). Exemplo:

  ```
  <!--#INCLUDE VIRTUAL="/docs/extra.txt" -->
  ```

<!--#EXEC *atributo*-->

Inclui no documento os valores retornados pela execução de um programa no servidor (provoca a execução desse programa). Suporta um atributo, que pode ser `CMD` ou `CGI`.

Atributos:

- `CMD="comando"`. Informa o comando do sistema a ser executado.
- `CGI="/cgi-bin/programa.pl"`. Informa, a partir do diretório raiz do servidor, o caminho (diretório CGI) e o nome de um programa CGI a ser executado. *Exemplo:*

  ```
  <!--#EXEC CGI="/cgi-bin/counter.pl?doc=includes.html"-->
  ```

  aciona o programa `counter.pl` que conta o número de vezes que o documento foi acessado:

<!--#FSIZE *atributo*-->

Inclui a informação sobre o tamanho de um determinado arquivo no documento. Suporta um atributo, que pode ser `FILE` ou `VIRTUAL`.

Atributos:

- `FILE="arquivo"`. Informa a localização de arquivo no sistema de arquivos do sistema operacional local (caminho relativo ou absoluto).

```
<!--#FSIZE FILE="c:\paginas\htdocs\img\imagem.gif" -->
```

- `VIRTUAL="arquivo"`. Informa a localização de arquivo no sistema de arquivos do servidor (caminho relativo ou absoluto). Exemplo:

```
<!--#FSIZE VIRTUAL="/img/imagem.gif" -->
```

<!--#FLASTMOD *atributo*-->

Inclui a informação sobre a data da última modificação de um determinado arquivo no documento. Suporta um atributo, que pode ser `FILE` ou `VIRTUAL`.

Atributos:

- `FILE="arquivo"`. Informa a localização de arquivo no sistema de arquivos do sistema operacional local (caminho relativo ou absoluto).
- `VIRTUAL="arquivo"`. Informa a localização de arquivo no sistema de arquivos do servidor (caminho relativo ou absoluto).

<!--#CONFIG *atributo* **-->**

Modifica vários aspectos dos dados importados por SSI.

Atributos

- `errmsg="mensagem de erro".` Muda o formato da mensagem de erro padrão.
- `sizefmt="bytes | abbrev"`. Muda o formato de exibição dos tamanhos de arquivo. Pode ser abreviado (MB, kB, etc.) ou em bytes.
- `timefmt="código de data/hora"`. Muda o formato de exibição de datas.

10.5. Exemplos

Nas seções a seguir, apresentamos alguns exemplos e aplicações de SSI em documentos HTML. A seção a seguir contém uma página de testes que você poderá usar para saber quais os comandos SSI que são suportados pelo seu servidor.

Página de testes

O programa abaixo (disponível no disquete que acompanha esta apostila) testa o suporte dos comandos SSI listados acima no seu servidor. Ele tentará incluir arquivos, executar programas e imprimir diversas variáveis de ambiente.

```
<HTML>
<HEAD>
<TITLE>Teste de Server-Side Includes</TITLE>
</HEAD>

<body bgcolor="#FFFFFF">
```

```
<H1>Server-Side Includes</H1>

<P>Variáveis de configuração SSI (observe que comentários podem ser usados
dentro dos descritores):
     <PRE>&lt;!--#config timefmt="%c" formato simples de data/hora --&gt;
     <!--#config timefmt="%c" formato simples de data/hora -->
     &lt;!--#config sizefmt="%d bytes"--&gt;
     <!--#config sizefmt="%d bytes"-->
     &lt;!--#config errmsg="##ERRO!##"--&gt;
     <!--#config errmsg="##ERRO!##"--></PRE>

<p>As diretivas SSI abaixo estão executadas ao lado. Para ver o arquivo original
     é necessário visualizá-lo no seu diretório e não através do servidor.
     <p><b>Configurações necessárias:</b>
     1) Para testar a diretiva EXEC é necessário informar o nome um arquivo CGI
        que retorne texto e colocá-lo no lugar de "progcgi.exe". 2) Para testar
        as diretivas INCLUDE, FSIZE e FLASTMOD é preciso substituir ssitext.txt
        e ssifile.ext por arquivos existentes no mesmo diretório onde está a
        página SHTML.

<PRE><B>DIRETIVA SSI                                          RESULTADO</B>
<HR>
&lt;!--#exec cgi="/cgi-bin/progcgi.exe"--&gt;  <!--#exec cgi="/cgi-bin/ssicgi.exe"-->
&lt;!--#include file="ssitext.txt"--&gt;   <br><!--#include file="ssitext.txt"-->
&lt;!--#fsize file="ssifile.txt"--&gt;         <!--#fsize file="ssifile.txt"-->
&lt;!--#flastmod file="ssifile.txt"--&gt;      <!--#flastmod file="ssifile.txt"-->
&lt;!--#echo var="DOCUMENT_NAME"--&gt;         <!--#echo var="DOCUMENT_NAME"-->
&lt;!--#echo var="DOCUMENT_URI"--&gt;          <!--#echo var="DOCUMENT_URI"-->
&lt;!--#echo var="LAST_MODIFIED"--&gt;         <!--#echo var="LAST_MODIFIED"-->
&lt;!--#echo var="QUERY_STRING"--&gt;          <!--#echo var="QUERY_STRING"-->
&lt;!--#echo var="QUERY_STRING_UNESCAPED"--&gt;<!--#echo var="QUERY_STRING_UNESCAPED"-->
&lt;!--#echo var="DATE_LOCAL"--&gt;            <!--#echo var="DATE_LOCAL"--&gt;
&lt;!--#echo var="DATE_GMT"--&gt;              <!--#echo var="DATE_GMT"-->
&lt;!--#echo var="SERVER_SOFTWARE"--&gt;       <!--#echo var="SERVER_SOFTWARE"-->
&lt;!--#echo var="SERVER_NAME"--&gt;           <!--#echo var="SERVER_NAME"-->
&lt;!--#echo var="SERVER_PROTOCOL"--&gt;       <!--#echo var="SERVER_PROTOCOL"-->
&lt;!--#echo var="REQUEST_METHOD"--&gt;        <!--#echo var="REQUEST_METHOD"-->
&lt;!--#echo var="REMOTE_HOST"--&gt;           <!--#echo var="REMOTE_HOST"-->
&lt;!--#echo var="HTTP_ACCEPT"--&gt;           <!--#echo var="HTTP_ACCEPT"-->
&lt;!--#echo var="HTTP_USER_AGENT"--&gt;       <!--#echo var="HTTP_USER_AGENT"-->
&lt;!--#echo var="REFERER"--&gt;               <!--#echo var="REFERER"-->
&lt;!--#echo var="BOGUS"--&gt;                 <!--#echo var="BOGUS"--></PRE>
<HR>
</BODY>
</HTML>
```

Data de modificação

Uma das aplicações mais comuns de SSI, nos servidores que suportem o comando ECHO é a impressão da data de modificação do arquivo:

```
<p>Este arquivo foi modificado em <!--#echo var="LAST_MODIFIED"-->
```

O usuário que recebe a página HTML não tem como saber se houve no lugar onde aparece a data um comando SSI, pois o código ostra o texto resultante da transformação:

> Este arquivo foi modificado em Friday, Aug 13, 1999 23:59:45 GMT-03

Repositórios de imagens ou outros arquivos

SSI é útil para criar páginas que oferecem um repositório de imagens e programas para download. Você pode informar a data de alteração e o tamanho do arquivo automaticamente usando os comandos FSIZE e FLASTMOD:

```
<p><a href="/imagens/grande.jpg">Foto 1</a>
(<!--#fsize virtual="/imagens/grande.jpg "-->).
Modificada em <!--#flastmod virtual="/imagens/grande.jpg"-->
```

No browser, o texto aparecerá como:

> Foto 1 (25.5 kB). Modificada em Friday, Aug 6, 1999 12:20:45 GMT-03

Usando o comando CONFIG é possível mudar o formato das datas, para que possam ser usadas internacionalmente.

Cabeçalhos e rodapés

Talvez a maior utilidade de SSI seja para criar *templates*, ou seja, estruturas que são adicionados a todas as páginas. Isto pode ser uma barra de navegação, um rodapé padrão, um logotipo, etc. Considere um arquivo de textos contendo o seguinte código HTML, que é um menu de navegação que deve aparecer em cima e embaixo de todos os arquivos:

```
<table width=100% cellspacing=0 cellpadding=0 border=0>
<tr>
<td align=left><a href="javascript: viaMail()">
   <img src="email.gif" border="0"></a></td>
<td align=center><a href="javascript: marcador()">
   <img src="marcar.gif" border="0"></a></td>
<td align=center><A HREF="notas.html">
   <img src="notas.gif" border="0"></A></td>
<td align=right><A HREF="indice.html">
   <img src="indice.gif" border="0"></A></td>
</tr></table><p></P>
```

Ele pode ser incluído em todos os arquivos do site sem que o código precise ser repetido. Isto pode ser feito com um simples `<!--#include -->`:

```
(...) <BODY><TABLE WIDTH=450><TR><TD>
<!--#include virtual="menu.txt" -->

<H1>Capítulo X (...)
```

O resultado agora é que todas as páginas que possuírem o INCLUDE terão o mesmo menu no local onde antes estava o comando. Se esse menu precisar ser alterado, ele será alterado em um único lugar, e automaticamente modificará todas as páginas.

10.6. Exercícios

1. Escreva uma página HTML que inclua um rodapé padrão contendo informações sobre autoria e a última modificação do arquivo usando SSI.
2. Escreva uma página HTML que inclua o número gerado em arquivo pelo programa contador counter.pl (é necessário configurar suporte ao descritor exec) em uma área da página. O arquivo é um CGI e está disponível no disquete que acompanha esta apostila. Poderá não ser necessário instalá-lo. Verifique com seu instrutor se o programa está disponível em alguma área de CGI do servidor.
3. Escreva uma página HTML com uma tabela vazia que seja capaz de incluir linhas adicionais através de um arquivo de texto carregado via SSI. O arquivo de textos deve conter linhas de tabela da forma:

```
<tr><td>... texto ...</td><td>... texto... </td></tr>
<tr><td>... texto ...</td><td>... texto... </td></tr>
...
```

11. Cookies

A TECNOLOGIA CONHECIDA COMO HTTP Cookies, surgiu em 1995 como um recurso proprietário do browser Netscape, que permitia que programas CGI gravassem informações em um arquivo de textos controlado pelo browser na máquina do cliente. Por oferecer uma solução simples para resolver uma das maiores limitações do HTTP – a incapacidade de preservar o estado das propriedades dos documentos em uma mesma sessão – os cookies logo passaram a ser suportados em outros browsers e por linguagens e tecnologias de suporte a operações no cliente e servidor. Cookies são um padrão da Internet e sua especificação está publicada em um RFC.

Um cookie não é um programa de computador, portanto não pode conter um vírus executável ou qualquer outro tipo de conteúdo ativo. Pode ocupar no máximo 4 kB de espaço no computador do cliente. Um servidor pode definir no máximo 20 cookies por domínio (endereço de rede) e o browser pode armazenar no máximo 300 cookies. Estas restrições referem-se ao browser *Netscape* e podem ser diferentes em outros browsers.

Há várias formas de manipular cookies:

- Através de CGI ou outra tecnologia de servidor, como LiveWire, ASP, JSP, PHP ou Servlets, pode-se criar ou recuperar cookies.
- Através de JavaScript também pode-se criar ou recuperar cookies.
- Através do descritor <META> em HTML, pode-se criar novos cookies ou redefinir cookies existentes, mas não recuperá-los.

Um cookie é enviado para um cliente no cabeçalho HTTP de uma resposta do servidor. Além da informação útil do cookie, que consiste de um par nome/valor, o servidor também inclui um informações sobre o domínio e caminho do sistema de arquivos onde o cookie é válido, o seu tempo de validade e indicações se ele deve ou não ser criptografado.

11.1. Criação de cookies via cabeçalhos HTTP

Cookies podem ser criados através de um cabeçalho HTTP usando CGI. Como vimos nos capítulos anteriores, parte ou todo o bloco de cabeçalhos pode ser gerado por um programa CGI ou equivalente. Quando um programa CGI gera um cabeçalho não precisa se restringir ao `Content-type`. Pode também incluir campos de informação sobre a página que o servidor não inclui por *default*. Para criar cookies, por exemplo, o programa deverá formar o servidor a definir um ou mais cabeçalhos `Set-Cookie`, que irão instruir ao browser para que guarde a informação passada em cookies. Suponha que um servidor gere o seguinte cabeçalho:

```
HTTP/1.0 200 OK
Date: Friday, June 13, 1997
Server: Apache 1.02
Set-Cookie: cliente=jan0017
Set-Cookie: nomeclt=Marie
Content-type: text/html
```

```
<HTML><HEAD>
<TITLE> Capitulo 11</TITLE>
(...)
```

Quando receber a resposta do servidor e interpretar os cabeçalhos acima, o browser irá gravar dois novos cookies na memória, contendo as informações `cliente=jan0017` e `nomeclt=Marie`. Essas informações poderão ser recuperadas em qualquer página que tenha origem no servidor que definiu os cookies enquanto a presente sessão do browser ainda estiver aberta.

Um cookie tem um escopo de validade que é limitado pelo tempo, pelo domínio (nome da máquina ou rede) que gerou o cabeçalho para criá-lo, pelo caminho do sistema de arquivos onde reside o programa que o criou e pelo browser que realizou a requisição. Em outro browser, em outro caminho ou em outro domínio o cookie não existe. Ele também não existe se o seu período de vida chegar ao fim. No cookie acima, o tempo de vida é limitado à sessão do browser, mas isto pode ser alterado.

11.2. Alteração do tempo de vida

Um cookie criado da forma mostrada na seção anterior pode ser lido por outros programas ou páginas enquanto o browser não for fechado. Se o usuário fechar o browser e entrar numa página que tentar ler o cookie, ele não existirá mais. Mas é possível fazer com que ele persista, criando cookies persistentes. Se após o par nome/valor, o cabeçalho Set-Cookie tiver um parâmetro `expires` seguida por uma data no futuro, as informações do cookie serão gravadas em arquivo (e não apenas na memória RAM) e poderão persistir além da sessão atual do browser:

```
Set-Cookie: nomeclt=Marie; expires=Monday, 15-Jan-99 13:02:55 GMT
```

11.3. Espaço de nomes de um Cookie

Várias páginas de um site podem definir cookies. O espaço de nomes de um cookie é determinado através de seu domínio e caminho. Em um mesmo espaço de nomes, só pode haver um cookie com um determinado nome. A definição de um cookie de mesmo nome que um cookie já existente no mesmo espaço, sobrepõe o cookie antigo.

Por *default*, o espaço de nomes de um cookie é o domínio e caminho atual onde foi criado. Para fazer o cookie valer em outro domínio, mais restritivo ou pertencente à mesma rede, é preciso definir o campo `domain`. Por exemplo, se o domínio de um cookie é `chocolate.biscoitos.com`, ele só pode ser lido na máquina `chocolate.biscoitos.com` mas não em `agua.biscoitos.com` e. Para ampliar seu alcance à máquina `agua.biscoitos.com` e a outras máquinas do domínio, o campo `domain` deve ser especificado da forma:

```
Set-Cookie: nome=valor; domain=.biscoitos.com
```

Somente máquinas dentro do domínio `.biscoitos.com` podem redefinir o domínio. Eles não podem ser alterados para abranger domínios externos.

O caminho dentro do domínio onde o cookie é válido é o mesmo caminho onde foi criado. O caminho pode ser alterado de forma que tenha um valor mais amplo definindo o campo `path`. Por exemplo, se um cookie foi criado em um diretório `/bolachas/`, ele só será lido por páginas ou programas que estejam armazenados em `/bolachas/`. Programas ou páginas em / ou em /cgi-bin/, por exemplo, não conseguirão enxergá-lo. Para que seu escopo seja ampliado para /, o que o tornaria acessível em todo o site, é necessário que um caminho novo seja especificado da forma:

```
Set-Cookie: nome=valor; path=/
```

Um cookie chamado `bis` definido em `/` antes não colidia com um cookie também chamado `bis` definido em `/bolachas/`. Agora sim e o resultado pode ser imprevisível. É, portanto, preciso ter cuidado ao ampliar o escopo de cookies. Não há como isso afetar outros usuários, mas o mesmo usuário, se for visitar outra parte do site, poderá ter problemas.

Um cookie pode ser apagado se for definido um novo cookie com o mesmo nome e caminho/domínio e com data de vencimento (campo `expires`) no passado.

11.4. Parâmetros

É possível combinar todos os parâmetros. A ordem não importa. Todos são separados por ponto e vírgula e têm a forma propriedade=valor. A sintaxe completa do cabeçalho `Set-Cookie` está mostrada abaixo. Os campos são separados por ponto-e-vírgula. Todos, exceto o primeiro campo que define o nome do cookie, são opcionais.

```
Set-Cookie: nome_do_cookie=valor_do_cookie;
            expires=data no formato GMT (RFC 850);
            domain=domínio onde o cookie é válido;
            path=caminho dentro do domínio onde o cookie é válido;
            secure
```

Os parâmetros do cabeçalho `Set-Cookie` são usados na definição de cookies tanto em programas CGI quanto em roteiros JavaScript, rodando no cliente. O significado de cada parâmetro está explicado na tabela abaixo:

Parâmetro	Descrição
nome=valor	*Este campo é obrigatório.* Seqüência de caracteres que não incluem acentos, ponto-e-vírgula, percentagem, vírgula ou espaço em branco. Para incluir esses caracteres é preciso usar um formato de codificação estilo URL. Em *JavaScript*, a função `escape()` codifica informações nesse formato e a função `unescape()` as decodifica.

`expires=`*data*	*Opcional.* Se presente, define uma data com o período de validade do cookie. Após esta data, o cookie deixará de existir. Se este campo não estiver presente, o cookie só existe enquanto durar a sessão do browser. A data deve estar no seguinte formato:
	`DiaDaSemana, dd-mes-aaaa hh:mm:ss GMT`
	Por exemplo:
	`Monday, 15-Jan-1999 13:02:55 GMT`

`domain=`*domínio*	*Opcional.* Se presente, define um *domínio* onde o cookie atual é válido. Se este campo não existir, o cookie será válido somente no domínio onde o cookie foi criado.
`path=`*caminho*	*Opcional.* Se presente, define o *caminho* onde um cookie é válido em um domínio. Se este campo não existir, será usado o caminho do documento que criou o cookie.
`secure`	*Opcional.* Se presente, impede que o cookie seja transmitido a menos que a transmissão seja segura (baseada em SSL ou SHTTP).

11.5. Criação de cookies via HTML

Um cookie pode ser criado através de HTML usando o descritor <META> e seu atributo HTTP-EQUIV. O atributo HTTP-EQUIV deve conter um cabeçalho HTTP. O valor do cabeçalho deve estar presente

no seu atributo CONTENT. A presença do um descritor <META> dentro de um bloco <HEAD> de uma página HTML, criará um cookie no cliente quando este for interpretar a página.

```
<HEAD>
<META HTTP-EQUIV="Set-Cookie"
      CONTENT="nomeclt=Marie; expires=Monday, 15-Jan-1999 13:02:55 GMT">
(...)
</HEAD>
```

11.6. Recuperação de cookies

Toda requisição de um browser ao servidor consiste de uma linha que contém o método de requisição, URL destino e protocolo, seguida de várias linhas de cabeçalho. É através de cabeçalhos que o cliente passa informações ao servidor, como, por exemplo, o nome do browser que enviou o pedido. Uma requisição HTTP típica (capítulo 3) tem a forma:

```
GET /index.html HTTP/1.0
User-Agent: Mozilla/4.5 (WinNT; I) [en]
Host: www.alnitak.org.br
Accept: image/gif, image/jpeg, */*
```

Quando um cookie é recuperado pelo browser, ele é enviado em todas as requisições à URLs que fazem parte do seu espaço de nomes, através do cabeçalho do cliente `Cookie`. Apenas o par nome/valor é armazenado no cabeçalho. As informações dos campos `expires`, `path`, e `domain` não aparecem:

```
Cookie: cliente=jan0017; nomeclt=Marie
```

O servidor pode recuperar as informações do cookie através do cabeçalho ou através da variável de ambiente `HTTP_COOKIE`, definida quando o servidor recebe uma requisição com o cabeçalho Cookie. Para fazer uso dos dados de `HTTP_COOKIE`, é preciso tratar a string que a variável contém, separando cada cookie pelo ponto-e-vírgula e identificando nome e valor através do sinal de igualdade.

Cookies são armazenados em ASCII e contém vários caracteres reservados, entre eles o ponto-e-vírgula e o sinal de igualdade. Para evitar problemas, o cookie deve ser armazenado com esses caracteres convertidos em formatos codificados (x-www-form-encoding, por exemplo). Na recuperação dos dados, será necessário fazer uma decodificação. Várias linguagens oferecem ferramentas para esse tipo de codificação. Exemplos são Perl, C e JavaScript.

11.7. Programas CGI que criam cookies

Para manipular com cookies em Perl pode-se usar o módulo CGI:: ou bibliotecas semelhantes à `cgi-lib.pl`. Você pode usar a biblioteca `cookie.pl` (disponível no site do curso) para criar e apagar cookies em programas em Perl. Transfira-a para o seu diretório `cgi-bin` e carregue-a como foi feito com a biblioteca cgi-lib.pl:

```
require "cookie.pl";
require "cgi-lib.pl";
```

Agora você pode usar as funções `&MakeCookieHeader` para fazer o cabeçalho que cria cookies (antes de terminar o cabeçalho com uma linha em branco) e `&ReadCookies`, que funciona como o `&ReadParse` do `cgi-lib.pl` para ler os cookies criados.

`&MakeCookieHeader` cria cabeçalho de cookie (e conseqüentemente, cria um cookie no browser). Pode ser chamada várias vezes (para criar vários cookies) mas sempre *antes* de imprimir o fim do cabeçalho (antes do `'print &PrintHeader'` da biblioteca `cgi-lib.pl` ou antes de *qualquer* linha em branco ("`xxx xxx \n\n`"). Deve ser chamada como o `&PrintHeader`, precedida de print.

A função deve ter no mínimo dois argumentos. A sintaxe geral é:

```
print &MakeCookieHeader("nome", "valor", vida_em_dias,
                        "caminho", "dominio", seguranca);
```

Os dois primeiros campos (obrigatórios) contém o nome e valor do cookie (que podem conter espaços, etc. Todos os outros campos são opcionais.

O tempo de vida é um número de ponto-flutuante que representa o número de dias que o cookie vai durar. Só o número for negativo, será criado um anti-cookie (que dizimará qualquer outro cookie do mesmo nome que ele encontrar). É um campo opcional. Se não estiver presente, o cookie durará apenas até o fim da sessão.

O quarto campo informa o caminho do diretório abaixo do qual o cookie é válido. Se ausente, o cookie valerá apenas no diretório atual. Este campo é obrigatório para cookies persistentes ou não que seja criados em um lugar (por exemplo, o diretório `cgi-bin/`) e lidos em outro lugar (por exemplo, através de JavaScript em um diretório de documentos qualquer como o "`/`"). Se o cookie não for persistente, deve-se usar `'undef'` no lugar do número de dias.

O penúltimo campo informa o domínio abaixo do qual o cookie é válido. Deve ter no mínimo dois pontos (pode começar por ponto, por exemplo: .abc.com). Os campos anteriores, se não definidos, deverão conter o valor `'undef'`. Não é feita verificação de domínio (ou caminho), portanto, tenha cuidado em informar um domínio válido.

O último campo, se presente, deve conter um número positivo, e indica que o cookie só deve ser criado ou enviado em uma conexão segura. Se a conexão não for segura, ele não é criado ou enviado para o servidor pelo browser (mesmo que o domínio e caminho coincidam).

Veja alguns exemplos de criação de cookies. Cookie temporario:

```
print &MakeCookieHeader("usuario", "josé");
```

Cookie persistente que durará 15 dias:

```
print &MakeCookieHeader("usuario", "joao", 15);
```

Cookie com duração de 15 dias, no domínio abc.com.br, no caminho / e seguro:

```
print &MakeCookieHeader("usuario", "joao", 15, "/",
                        ".abc.com.br", 1);
```

Parâmetros não utilizados devem ser preenchidos com `undef`:

```
print &MakeCookieHeader("usuario", "joao", undef,
                         undef, ".abc.com.br");

print &MakeCookieHeader("usuario", "joao", undef, "/");
```

Lembre-se que todos os comandos acima devem ocorrer antes de qualquer linha em branco (pois ela terminará o cabeçalho).

11.8. Programas CGI que recuperam cookies

Para recuperar os cookies, eles devem estar no mesmo caminho e domínio e o browser que os lê também deve ser o mesmo. A biblioteca cookie.pl oferece a função `&ReadCookies` que funciona da mesma forma que

`&ReadParse` do `cgi-lib`: `&ReadCookies` lê todos os cookies e os coloca em vetor associativo escolhido pelo programador:

```
&ReadCookies(\%crackers);
```

Agora é só extrair o cookie, informando o seu nome no vetor associativo recém criado:

```
$cookie1 = $crackers{'nome1'}; # obtem o valor, informando-se o nome
$cookie2 = $crackers{'nome2'};
```

11.9. Exercícios

1. Crie um programa CGI (assacookie.pl) que ofereça um formulário HTML que pede o nome do usuário. Grave o nome do usuário como um cookie. Em outra página (comecookie.pl), leia a variável de ambiente e imprima na página o nome do usuário com uma mensagem de boas vindas.
2. Crie uma página HTML, com três botões de rádio (<INPUT TYPE=RADIO>). A página deve enviar seus dados para o programa computa.pl que primeiro tenta ler um cookie chamado "num". Se "num" não existir, ele deverá gravar um "cookie" num com o valor 1. Se "num" existe, ele deve somar 1 ao valor de "num" e gravar um novo cookie com a soma. Depois, deve gravar outro cookie "opcao" com a opção escolhida. O programa computa.pl deverá gravar os cookies e depois imprimir uma linha de cabeçalho "Location: http://URL" e terminar o bloco de cabeçalho com uma linha em branco (não deverá ter "Content-type"). Location redireciona a página. A URL deverá ser o da página que contém os botões. Na quinta vez em que o usuário clicar os botões, o programa não devera redirecioná-lo para outra página, mas terminar o cabeçalho com "Content-type" e gerar uma página HTML informando as opções que o usuário fez.
3. Faça um contador de visitas para que um usuário saiba quantas vezes ele já visitou sua página. O cookie deve ser persistente para que não desapareça quando o usuário fechar o browser. Faça-o durar 5 dias.

12. Princípios de programação

O objetivo deste capítulo é oferecer uma introdução aos princípios de programação de computadores que será necessária para o módulo seguinte.

Para programar é preciso saber uma linguagem de programação. Mas para aprender tal linguagem, é preciso não só praticar, mas também conhecer como funciona um programa e como o código de um programa é estruturado. A maior parte das linguagens de programação usadas hoje em dia tem estruturas semelhantes, que são usadas para controlar *quando* e *como* as instruções serão executadas. Os conceitos aqui apresentados têm por finalidade ajudar àqueles que nunca programaram a identificar as principais partes de um programa e ganhar alguma experiência na análise de programas, para que sejam capazes de aprender JavaScript – apresentada no módulo seguinte – e alterar ou criar programas em Perl (se desejar) para utilizar em seu site.

Perl não é objeto central deste curso e está apresentada nesta apostila em um apêndice.

12.1. Tipos de linguagens

Linguagens de computador podem ser classificadas de várias formas. Em um capítulo anterior as classificamos em gerações, que descreviam o quanto tais linguagens se distanciavam da linguagem da máquina e se aproximavam da linguagem humana.

Todas as linguagens contém *instruções*, que podem ser simples ou compostas por várias outras instruções mais simples. Nas linguagens de *alto-nível* (mais distantes da linguagem de máquina) essas instruções são geralmente expressas em linguagem humana (ou algo parecido). Uma instrução geralmente sugere *ação*, e *verbos* formam a maior parte das instruções das linguagens que são usadas para construir *procedimentos*. Por exemplo, um programa que lê um texto e o traduz para outra língua provavelmente terá instruções como "leia", "compare", "altere", expressas no vocabulário específico da linguagem. Existem também linguagens que não contém verbos, mas *substantivos*. HTML é um exemplo. Nesse caso, o interpretador já sabe quais ações irá tomar, mas depende da linguagem para saber *como* deverá proceder.

Linguagens procedurais

As linguagens que contém instruções capazes de expressar ações possuem a capacidade de construir procedimentos ou programas. Linguagens que são totalmente dependentes das instruções de ação são chamadas de *linguagens procedurais*. Exemplos são os programas em Shell que construímos em capítulos anteriores, que consistiam de uma lista de comandos Unix. Outras linguagens procedurais são C, Pascal, FORTRAN, BASIC, COBOL e Perl.

Linguagens declarativas

Linguagens que apenas declaram *como* um interpretador deve agir são *linguagens declarativas*. Não é possível construir programas apenas usando linguagens declarativas. Com HTML você pode descrever totalmente um texto. Com XML você pode descrever qualquer tipo de dados ou informação. Com CSS você pode descrever a cor e o tamanho de um parágrafo, mas quem faz o texto mudar de cor é o interpretador embutido no navegador. Não é possível com essas linguagens criar novos comportamentos que o navegador não tenha previsto.

Linguagens baseadas ou orientadas a objetos

As linguagens modernas raramente são 100% procedurais. Ela geralmente combinam estruturas com procedimentos, tratando as estruturas declaradas como *objetos*, e oferecendo instruções de ação que permitem manipular não só valores individuais como números e palavras, mas estruturas inteiras, que podem consistir de coleções de números e palavras. JavaScript, por exemplo, quando executado por um navegador, comporta-se como uma *linguagem baseada em objetos*. É possível, através de instruções da linguagem, ligar o evento de se mover um mouse sobre uma imagem à mudança da cor do texto de um parágrafo. CSS define a cor do parágrafo, que é formado de texto identificado por HTML. O browser, ao ler HTML e CSS, define sua cor e tamanho iniciais e depois mantém a página estática. Instruções de JavaScript, portanto, introduzem um novo comportamento (não previsto no browser), que podem identificar o objeto (pela estrutura do HTML) e alterar seus atributos (declarados em CSS).

Há várias outras linguagens baseadas ou orientadas a objetos. A maioria têm sua própria sintaxe e seus próprios mecanismos para identificar, manipular e operar com objetos. Exemplos são C++, Delphi, Java, Visual Basic e Perl 5.

12.2. Componentes de um programa

É comum usar um programa para realizar um determinado cálculo e, a partir dos resultados obtidos, realizar uma ou outra operação. Outra tarefa freqüente é a repetição. Pode-se criar um programa para repetir um certo procedimento cem vezes ou até que certa condição tenha sido atingida. Nas linguagens antigas, cada instrução ocorria em uma linha numerada. Para repetir uma seqüência de comandos, precisava-se saber qual a linha onde a seqüência começava e escrever uma instrução que pulasse àquela linha.

Nas linguagens modernas, a realização de operações é feita de forma mais ou menos padronizada, e não depende mais de linhas numeradas. Todas as linguagens modernas (chamadas de estruturadas) possuem estruturas usadas para controlar o fluxo de controle do programa. Uma seqüência de instruções pode ser agrupadas em um *bloco* e chamado como um roteiro externo ou *subrotina*. Valores podem ser armazenados em *variáveis*. Coleções de valores podem ser armazenadas em *estruturas de dados* e repetições e decisões podem ser realizadas usando *estruturas de controle de fluxo*. Todos esses termos, presentes em linguagens procedurais e linguagens baseadas em objetos), serão definidos abaixo.

Algoritmo: fluxo de controle

Qualquer procedimento segue uma seqüência de instruções que tem início e fim. Essa seqüência é chamada de algoritmo. Entre o início e o fim pode haver instruções que forcem a repetição de trechos do procedimento, que façam cálculos, que tomem diferentes decisões baseadas nesses cálculos ou até que forcem a saída do procedimento antes do seu término.

Um algoritmo pode ser tão simples a ponto de apenas ter instruções que são executadas em seqüência, com um resultado final previsível; ou ser complicado a ponto de ter milhares de resultados possíveis, dependendo de condições internas e externas, e apresentar resultados difíceis de se prever. A complexidade de um algoritmo às vezes é desnecessária. Algoritmos complexos não só são difíceis de analisar (e consertar, quando há erros), mas

também afetam o desempenho do computador e, devido à sua imprevisibilidade, podem oferecer riscos à segurança do programa.

Variáveis

Variáveis são abstrações da memória de um computador e permitem o armazenamento de valores temporários usados para realizar cálculos, transformações em textos, etc. Dependendo do que se armazena em uma variável, o espaço em memória (em bits) ocupado poderá ser maior ou menor. A forma como os dados armazenados são interpretados também são importantes. Um número armazenado pode ser um número inteiro ou apenas a parte do número que segue depois da vírgula. Pode também ser apenas um código de cor, ou de caractere, ou ainda ter dígitos que identificam seu sinal.

Por causa das diferenças em tamanho e forma, toda variável precisa não só identificar o valor que contém, mas também o seu *tipo*. Através do tipo, o programa sabe quanto espaço a variável requer e como as informações armazenadas devem ser interpretadas. Tipos de dados comuns são *número* (inteiro de vários tamanhos ou de ponto flutuante), *caractere* (número sem sinal que representa uma letra ou símbolo) ou *estado booleano* (indicador de estado verdadeiro ou falso).

Esses tipos são também chamados de *tipos primitivos*, pois são fundamentais em uma linguagem e não podem ser divididos em partes menores. Outros tipos de dados como datas, por exemplo, são *complexos* (uma data pode ser dividida em três números, representando dia, mês e ano). Tipos complexos, em linguagens orientadas a objetos, são definidos através de estruturas de dados chamadas de *objetos*.

Em um programa, variáveis precisam receber seu valor através de uma operação de *atribuição*. A forma de fazer isto depende de cada linguagem. Veja alguns exemplos:

```
a = 5;        (Java, JavaScript e C)
a := 5;       (Delphi, Pascal)
$a = 5;       (Perl)
```

O nome da variável acima é "a". Esse nome é chamado de *identificador*. Em Perl os identificadores de variáveis precisam começar com um $. Nas outras linguagens acima pode ser um nome que não comece com número. Todas as instruções acima atribuem (armazenam) o valor 5 na variável "a".

Algumas linguagens, a partir da atribuição, automaticamente adivinham o tipo de dados armazenado da variável (por exemplo, se é número, se é texto, se é booleano). Outras exigem que, antes de qualquer atribuição, o tipo de dados da variável seja *declarado*. Isto ocorre em linguagens como Java ou C, mas não em JavaScript ou Perl:

```
int a;        (declara que "a" armazena valores numéricos inteiros)
a = 5;
```

O trecho acima é uma declaração Java ou C.

Subrotinas

Um programa consiste de uma seqüência de instruções. Freqüentemente, várias seqüências precisam ser repetidas mais de uma vez. Elas são completas e funcionam como pequenos programas. Por exemplo, considere um programa que calcula taxas de juros. Ele terá que várias vezes repetir uma seqüência de operações de soma e multiplicação. A seqüência de operações que realiza o somatório poderia ser considerado um programa em si. A seqüência pode então receber um nome e ser definida como *subrotina*, podendo então ser chamada (pelo nome) várias vezes dentro do programa principal.

Subrotinas são usadas em todas as principais linguagens de programação. As linguagens estruturadas as chamam de *funções*. As linguagens orientadas a objetos as chamam de *métodos*. Os termos não significam exatamen-

te a mesma coisa mas todos referem-se a seqüências de instruções, localizadas fora do programa principal, e que podem ser identificadas pelo nome.

Algumas subrotinas são simplesmente seqüências de instruções independentes de qualquer valor inicial e que não retornam qualquer valor como resultado. Algumas exigem parâmetros de entrada e podem devolver um valor como resultado. Um exemplo é uma subrotina que realiza uma soma. Ela precisa receber como entrada dois números, e no final devolve o resultado que é a soma deles. As instruções em seu interior nada interessam ao programa principal, apenas a sua entrada e saída.

Assim como as variáveis, cada linguagem tem a sua própria forma de representar subrotinas, funções ou métodos. Em Perl, as subrotinas são representadas por uma seqüência de instruções entre chaves, precedidas pelo seu nome (o identificador da subrotina):

```
sub imprime {
   print "Linha 1: Olá! \n";
   print "Linha 2: Tchau! \n";
}
```

A subrotina acima se chama imprime, e pode ser chamada várias vezes de um programa em Perl usando a notação:

```
&imprime;
```

Um & é usado em Perl para indicar que o nome que segue é identificador de uma subrotina que deve ser chamada. Um ponto-e-vírgula termina todas as instruções em Perl.

Subrotinas que retornam valores podem ter esse valor atribuído diretamente a uma variável:

```
$resultado = &soma(5, 6);
```

Os parênteses que seguem o identificador da subrotina são os seus parâmetros. A maior parte das linguagens usa essa notação ou uma notação semelhante para indicar os parâmetros. Em Perl, a chamada acima poderia ser usada para invocar a subrotina:

```
sub soma {
    ($a, $b) = @_;          (obtém os parâmetros passados e os copia em $a e $b)
    return $a + $b;
}
```

A instrução return devolve o valor resultante da expressão $a + $b.

Em JavaScript, uma subrotina (ou função) é definida da forma:

```
function soma(a, b) {
    return a + b;
}
```

e pode ser chamada da forma:

```
resultado = soma(5, 6);
```

Neste caso, 5 é copiado em a e 6 em b.

Em Java, C, Delphi, VB a sintaxe é outra, mas as formas de chamar, passar parâmetros e obter resultados retornados é parecida. Quando você aprende uma linguagem estruturada, o aprendizado de uma segunda ou terceira é bem mais fácil.

Exercício

1. Abra um programa (código-fonte) escrito em JavaScript ou em Perl. Procure e tente identificar as variáveis, chamadas e definições de funções e subrotinas.

Estruturas de dados e objetos

A maior parte das linguagens permite que se manipule não só com valores primitivos como números e texto, mas também com estruturas complexas. Nas linguagens baseadas em objetos ou orientadas a objetos, geralmente não é necessário criar tais estruturas, pois grande parte delas já está pronta.

Com estruturas, pode-se ter *tipos de dados abstratos* definidos como coleções de tipos primitivos. Essas coleções podem ser manipuladas através de uma única variável através da qual se possa extrair as propriedades do objeto.

Uma das estruturas mais simples é o *vetor*, que consiste de uma coleção ordenada de valores, geralmente (mas não sempre) do mesmo tipo. O vetor permite que seus elementos sejam recuperados através de um índice que indique a sua posição. Um vetor em Perl pode ser definido da seguinte maneira:

```
@meses = ('janeiro', 'fevereiro', 'março', 'abril', 'maio', 'junho');
```

@meses é o identificador. Identificadores que começam com @ em Perl são vetores. Mas @ é usado somente na definição. Para obter um dos elementos do vetor acima, é preciso saber sua posição. A contagem começa de 0:

```
$mes = $meses[3];
```

O índice é passado entre colchetes logo após o identificador do vetor, desta vez precedido por um $ (pois um elemento do vetor é um valor primitivo). O índice 3 corresponde ao valor "abril".

Em JavaScript, vetores são criados da seguinte maneira:

```
meses = new Array('janeiro', 'fevereiro', 'março', 'abril', 'maio', 'junho');
```

A forma de manipulação é a mesma (só não tem o @ nem o $):

```
mes = meses[3];
```

Outra estrutura comum é o *hash*, também chamado de *vetor associativo* ou *hashtable*. O hash consiste em coleções formadas por pares de valores. Um dos valores serve de chave para se obter o segundo. Por exemplo, pode-se ter um hash para armazenar cores HTML onde utilizando a palavra "azul" como chave, se obtenha o número 0000ff. Em Perl, Um hash pode ser definido através de um vetor atribuído a um identificador precedido de %:

```
%cores = ('vermelho', 'ff0000', 'verde', '00ff00', 'azul', '0000ff');
```

Na definição, os pares são ordenados mas isto não influi na forma de obter os resultados. A cor correspondente a verde pode ser obtida fazendo:

```
$codigo = $cores{'verde'};
```

Observe que o "%" só aparece quando o hash é definido. Quando é usado aparece com um "$" antes. Uma nova cor pode ser colocada no hash fazendo simplesmente a atribuição:

```
$cores{'magenta'} = 'ff00ff';
```

Em uma linguagem baseada em objetos como JavaScript, as estruturas definidas em uma página HTML podem ser manipuladas pelo programa como *objetos*. Objetos funcionam como coleções de valores que podem ser atribuídos a uma variável (como vetores e hashes), mas podem também conter coleções de instruções (métodos).

Em um objeto, os valores que armazena são suas propriedades. Em JavaScript, propriedades são identificadas (ou criadas) com identificador próprio, ligado ao identificador do objeto através do ponto "." O objeto cores abaixo está definindo três propriedades:

```
cores.verde = "00ff00";
cores.azul = "0000ff";
cores.vermelho = "ff0000";
```

Agora uma das propriedades foi copiada para uma variável:

```
cor = cores.verde;
```

A variável cor contém o valor "00ff00".

Em Perl 5, que é uma linguagem orientada a objetos, as propriedades de um objetos são acessadas através do "->" que funciona de forma idêntica ao ponto:

```
cores->azul = "0000ff";
```

Algumas propriedades de objetos realizam ações. São os métodos. Em JavaScript eles são fáceis de identificar pois sempre têm um par de parênteses após o identificador (como as funções), mesmo que não tenha argumentos:

```
document.write("<p>Linha 2");
hoje.toGMTString();
```

Nos exemplos acima, document e hoje são objetos. O primeiro é um objeto criado pelo browser e representa a página HTML.

Exercício

2. Abra programas em Perl e páginas HTML com JavaScript e tente identificar vetores, objetos, hashes, propriedades e métodos.

Operadores e expressões

Linguagens têm uma coleção de *operadores* com os quais se pode formar expressões e, com elas, instruções. Já vimos o importante operador de atribuição, que permite copiar um valor para uma variável. Todos os programas também possuem os operadores básicos de aritmética: soma (+), subtração (-), multiplicação (*) e divisão (/). Outros operadores como resto, expoentes, raiz quadrada, operadores de comparação, operadores booleanos, etc. podem estar presentes mas apresentar uma sintaxe diferente. Às vezes um operador não está presente na linguagem como símbolo, mas somente através de uma função (subrotina) que é chamada externamente.

Combinando valores, operadores e variáveis obtém-se *expressões*. Toda expressão possui um *tipo de dados* (geralmente o tipo de dados da variável à qual o valor final é atribuído). Expressões podem também ser combinadas, organizadas em estruturas de controle de fluxo (seção seguinte) de forma a elaborar algoritmos complexos.

As diferentes sintaxes entre linguagens às vezes podem confundir. Por exemplo, o símbolo "." (ponto), usado em JavaScript para referenciar propriedades de objetos é usado em Perl para concatenar texto:

```
$dia = "sexta";
$frase = "Hoje é" . $dia . "-feira!";
```

Em JavaScript, a concatenação é feita usando o mesmo operador usado na adição:

```
dia = "sexta";
frase = "Hoje é" + dia + "-feira!";
```

Ambas as linguagens usam o operador "=" exclusivamente para atribuição. Para comparar um valor com outro é utilizado o operador "==":

```
if (dia == "sexta") { ... }
```

Exercícios

3. Abra um programa em Perl (ou JavaScript) e identifique suas expressões e tente compreender o funcionamento de algumas de suas instruções.

Estruturas de controle de fluxo

Para controlar a forma como uma seqüência de instruções é executada, a maior parte das linguagens estruturadas conta com estruturas de controle de fluxo. Elas permitem a realização de repetições e tomada de decisões. Sempre afetam um bloco de instruções. As principais estruturas são as estruturas IF (se alguma condição for verdadeira execute o bloco) e WHILE (enquanto alguma condição for verdadeira execute o bloco), presentes em todas as linguagens.

A estrutura IF geralmente opera sobre uma expressão booleana (um teste) que resulta em um valor verdadeiro ou falso. Ela precede um bloco de instruções que só será executado se a expressão for verdadeira. Tanto em Perl como em JavaScript, o bloco é delimitado por chaves { e }, e o IF escrito em letras minúsculas, antes de uma condição entre parênteses:

```
if (indice > 16) {
  ... instruções ...
}
```

Nem todas as linguagens usam chaves para iniciar e fechar blocos. Umas usam as palavras BEGIN e END, outras usam o nome da estrutura ao contrário (IF e FI, WHILE e ELIHW).

O IF, portanto, serve para que um programa possa tomar decisões. Às vezes ele deseja fazer algo quando a expressão do IF não resultar verdadeira. Ele pode controlar isto usando os operadores da linguagem para negar a expressão ou usar uma estrutura ELSE (caso-contrário). ELSE complementa o IF:

```
if (indice > 16) {
  ... instruções que serão executadas se indice for maior que 16
} else {
  ... instruções executadas se indice for maior ou igual a 16.
}
```

O IF pode ser combinado com o ELSE para criar estruturas de decisão mais complexas usando IF-ELSE IF-ELSE. A estrutura abaixo é válida em Perl:

```
if ($indice > 16) {
  ... instruções que serão executadas se indice for maior que 16
} elsif ($indice <= 20) {
  ... instruções executadas se indice for menor ou igual a 20.
} else {
  ... executadas se indice for maior ou igual a 16 e menor que 20.
}
```

A estrutura WHILE é usada para realizar repetições. A cada repetição o programa testa sua condição. Se a condição continuar verdadeira, uma nova rodada é executada. Se for falsa, o programa deixa o bloco WHILE. É

preciso que a condição mude de alguma maneira ou a repetição jamais chegará ao fim. A sintaxe de WHILE é a mesma em Perl e JavaScript:

```
x = 10;
while (x >= 0) {
  ... instruções ...
  x = x - 2;
}
```

As instruções acima serão repetidas cinco vezes pois a cada repetição, a variável testada (inicializada com o valor 10) diminui (na expressão x = x – 2), até que chegará o momento em que a expressão do WHILE será falsa, e o controle deixará o bloco.

A expressão x = x – 2 é uma operação de decremento. O seu oposto é uma operação de incremento. Tais operações são muito comuns em programas e são essenciais para que possa existir controle de blocos de repetições. A combinação inicialização – teste – incremento ou decremento é tão comum que há uma estrutura que já embute as três operações em sua declaração para realizar repetições: a estrutura FOR.

A sintaxe de FOR em JavaScript e Perl é a mesma. As três operações aparecem separadas por pontos e vírgulas:

```
for (x = 10; x >= 0; x = x - 2) {
  ... instruções ...
}
```

O bloco acima faz o mesmo que o WHILE mostrado antes. A inicialização é feita apenas uma vez, o teste é executado cada vez que o bloco chega ao fim e o incremento ou decremento é executado logo após a última instrução do bloco.

As linguagens geralmente possuem outras estruturas além das mostradas, mas com estas três é possível escrever qualquer algoritmo. Conhecendo e sabendo identificar essas estruturas no código-fonte de um programa permitirá analisar e saber acompanhar qualquer programa.

Exercícios

4. Identifique as estruturas WHILE, FOR e IF em programas Perl e páginas HTML com JavaScript. Tente, através delas, prever o funcionamento do programa.

12.3. Como escrever um programa

Um programa geralmente pode ser escrito usando um editor de textos comum. Ao escrever programas em Perl ou JavaScript, use sempre um editor de textos que tenha a capacidade de salvar em formato texto (não use editores como o Word, por exemplo).

O código-fonte

O código-fonte é o programa propriamente dito, escrito numa linguagem de programação compreendida pelo programador, que tem as estruturas, expressões e componentes que descrevemos na seção anterior. Dependendo de como o programa é executado, o código-fonte pode ou não ser visível pelo usuário do programa.

O programa-objeto

O programa-objeto é o arquivo que contém o programa pronto para a execução. Se o programa foi escrito em uma linguagem interpretada, o programa-objeto e o programa-fonte são a mesma coisa. Será preciso que o

interpretador leia o código e o interprete durante a execução. Se o programa foi escrito em uma linguagem que exige uma etapa de compilação, o programa-objeto provavelmente contém linguagem de máquina, e poderá ser executado diretamente pelo sistema operacional sem precisar de interpretador. A maior parte dos softwares são distribuídos como programas-objeto compilados (você não tem acesso ao seu código-fonte).

A execução

A execução do programa, seja ele compilado ou interpretado, faz com que ele seja transferido para a memória temporária do computador e geralmente causa a criação de um processo no sistema operacional. É somente neste estado que o programa pode realizar as tarefas programadas.

WebDesigner / IBPINET

Criação de Web Sites II

Material de referência
Introdução à Linguagem Perl

Conteúdo

Introdução à linguagem Perl *132*

1. Sintaxe básica *133*

1.1. Instruções 134
1.2. Literais e identificadores 134
1.3. Operadores 135
1.4. Variáveis escalares 136
1.5. Vetores indexados 138
1.6. Variáveis de vetores 138
1.7. Operadores de vetores 139
1.8. Estruturas de controle 140
1.9. Outros operadores de controle de fluxo 141
1.10. Vetores associativos 143
1.11. Operadores de associações 143
1.12. Operador eval 143

2. Entrada e saída *145*

2.1. Entrada e saída padrão 145
2.2. Arquivos 147
2.3. Diretórios 149

3. Expressões regulares *151*

3.1. Sintaxe básica 151
3.2. Mudança de delimitadores 152
3.3. Variáveis 152
3.4. Padrões simples 153
3.5. Seqüências, multiplicadores e alternação 153
3.6. Âncoras 154
3.7. Parênteses como memória 155
3.8. Outros operadores 156
3.9. Substituições 157
3.10. Transliteração 157
3.11. Operadores split e join 159

4. Subrotinas e bibliotecas *160*

4.1. Subrotinas 160
4.2. Bibliotecas 161

5. Transformação de dados *163*

5.1. Manipulação de texto 163
5.2. Formatos 164

6. Módulos *166*

6.1. O que são módulos de classe 166
6.2. Como usar módulos 167
6.3. Módulos para uso com CGI 167
Exemplo de uso 167
Exemplo com File Upload [GUND96] 168
6.4. Módulo ODBC para Windows 169
Como usar o módulo Win32::ODBC 169
Principais métodos 169
CGI para acesso via ODBC 171
6.5. Módulo gráfico GD 172
Objeto Image 172
Objeto Polygon 173
Objeto Font 173
Construtor newFromGif (método de classe) 173
Principais métodos do objeto GD::Image 173
7. Referências 177

Introdução à linguagem Perl

ESTE APÊNDICE destina-se às pessoas que têm alguma familiaridade com linguagens de programação de computadores. A finalidade deste material é servir como referência para quem quiser se aprofundar um pouco mais nessa linguagem com o objetivo de desenvolver (ou alterar) programas CGI. O texto contém uma explicação da estrutura, sintaxe e recursos da linguagem seguida de exemplos com trechos de código. Apesar de abordar assuntos que fogem ao escopo do curso de Web Designer, esta introdução não é exaustiva e é insuficiente para quem tiver como objetivo conhecer Perl a fundo. Para saber utilizar recursos como controle de processos, ponteiros e construção de soquetes de rede (TCP/IP) você deve procurar documentação mais detalhada como referências listadas no final.

Tópicos abordados neste capítulo

- Introdução à linguagem *Perl*
- Sintaxe básica e estrutura
- Entrada e saída de dados via dispositivos padrões e arquivos
- Expressões regulares do *Unix* disponíveis em *Perl*
- Criação e utilização de funções
- Transformação de *strings*, formatos e outros tópicos avançados
- Uso elementar de módulos do *Perl* 5

Objetivos

No final deste capítulo você deverá ser capaz de:

- Conhecer a sintaxe básica da linguagem *Perl* e Utilizar os recursos disponíveis na linguagem para desenvolver programas utilitários utilizáveis em *Unix* e *Windows*.
- Analisar um programa simples em *Perl* (como um *script* CGI), identificar suas partes, entendê-lo de uma forma geral (saber o que ele faz) e saber alterá-lo (programas em *Perl 4* ou usando módulos apresentados aqui).
- Saber o que são expressões regulares e como usá-las.
- Incluir módulos e bibliotecas em um programa e conhecer a sintaxe básica utilizada para construir objetos e chamar seus métodos. Para usar os módulos você deverá consultar a documentação de cada um.

1. Sintaxe básica

De acordo com seu criador, Larry Wall, *Perl* é abreviação de *Practical Extraction and Report Language* (Linguagem para a Extração Prática e Relatórios) e também de *Pathological Ecletic Rubbish Lister* (Listador de Lixo Patologicamente Eclético). Foi criado para auxiliar o usuário e administrador *Unix* a realizar tarefas que são muito complicadas para o *Shell* (linguagem de propósito geral para automação de tarefas do *Unix*) mas ao mesmo tempo de vida muito curta ou complicada demais para escrever em *C* ou outra linguagem do *Unix*. Por causa dos seus recursos de formatação de texto e recursos disponíveis para o desenvolvimento de programas pequenos, porém sofisticados, *Perl* se tornou a linguagem preferencial dos desenvolvedores CGI.

Perl é uma linguagem interpretada. Para executar um programa em *Perl*, ele deve ser armazenado em um arquivo com extensão `.pl` e executado através da invocação do interpretador, da forma:

```
perl seuprog.pl
```

Em sistemas *Unix*, qualquer *script* pode ser transformado em um programa executável. Para isto, é preciso que o programa informe, na primeira linha do código, onde está o interpretador. O clássico e insuportável programa *Hello World* pode ser escrito em *Perl* da seguinte maneira:

```
#!/usr/local/bin/perl
print "Hello, world!\n";
```

A primeira linha é um comentário para a linguagem *Perl*. Em sistemas *Unix*, diz que este é um programa em *Perl* e informa o endereço do interpretador, o que permite que o programa seja executado diretamente sem precisar chamá-lo, na linha de comando, ou seja, pode-se fazer simplesmente:

```
./seuprog.pl
```

para executar o programa. É necessário que o bit executável do arquivo seja ligado para que isto funcione

```
chmod a+x seuprog.pl
```

Em sistemas *Windows*, o comentário da primeira linha geralmente é ignorado. Mas, em programas CGI, alguns servidores Web como o *Apache* utilizam a primeira linha no *Windows* para localizar o interpretador, por exemplo:

```
#c:\perl\bin\perl.exe
print "Hello, world!\n";
```

A maior parte dos servidores Web, porém, utiliza-se de outra forma de associação entre o interpretador e o programa.

No exemplo acima, a segunda linha é o código executável do programa. A instrução `print` imprime o seu argumento na *saída padrão*. No final do *string* que é argumento de `print`, há um código de escape especial: "`\n`". Este caractere provoca uma quebra de linha. Observe que a linha termina em ";". Isto ocorre em *todas* as instruções simples do *Perl*.

Para rodar o programa, basta salvá-lo (como "`hello.pl`", por exemplo) e rodá-lo:

```
perl hello.pl
```

ou diretamente,

```
./hello.pl
```

se o programa estiver em ambiente *Unix* com bit executável ligado e endereço do interpretador informado corretamente. O resultado será

```
Hello, world!
```

1.1. Instruções

A sintaxe das instruções em *Perl* é semelhante à de *C* (ou *Java*). Toda instrução simples termina com um "`;`" (ponto-e-vírgula). Instruções compostas podem ser agrupadas entre chaves (funções, estruturas `for`, `while`, etc.). O espaço em branco (tabulações, quebras de linha, espaços) é ignorado, exceto dentro de *strings*.

Para programar em *Perl*, pode-se utilizar o alfabeto ISO-Latin-1 (8 bits). Os identificadores, porém, devem limitar-se ao alfabeto ASCII.

Comentários valem até o fim de uma linha e são precedidos do caractere #.

1.2. Literais e identificadores

Literais são representações dos valores escalares utilizados para representar números ou *strings*. Pode-se transformar dados escalares (como somar, concatenar) e armazená-los em variáveis representadas por identificadores.

Todos os valores numéricos têm, internamente, o mesmo formato em *Perl*: ponto-flutuante de dupla precisão, embora possam ser representados de várias maneiras. Os literais numéricos que representam números de ponto-flutuante podem conter ponto decimal, sinal e a letra "`e`", indicando expoente (`12.3e-34`). Os que representam inteiros são por *default*, decimais. Podem representar valores octais se precedidos por "`0`" ou hexadecimais, se precedidos por "`0x`".

Strings são seqüências de caracteres. Cada caractere tem 8 bits (*Perl* 4). O menor *string* é o *string* vazio. O maior, ocupa toda a memória disponível. Como você pode manipular com *strings* de 8 bits, valores binários podem ser manipulados em *Perl* se forem tratados como *strings*.

Há três representações diferentes para *strings* em *Perl*. *Strings* entre apóstrofes (aspas simples) é uma seqüência de caracteres armazenada entre apóstrofes. Não há caracteres ilegais. Até o caractere de quebra de linha (se a linha quebrar dentro do *string*) é considerado parte dele. Tais *strings* podem, portanto, ser usados para até com dados binários.

```
$texto_longo = "
<HTML>
<HEAD><TITLE>A Divina Comédia - Inferno: Canto 1</TITLE></HEAD>
<BODY><TABLE WIDTH=450><TR><TD>
<H1>Canto I</H1>
<P>Quando eu me encontrava na metade do caminho de nossa vida, me vi
perdido em uma selva escura, e a minha vida não mais seguia o cami-
nho certo.
(...)
```

```
</TD></TR></TABLE></BODY></HTML>
"; # este string não é neutro (seria se estivesse entre ' e ')
```

Alguns caracteres, quando aparecem dentro de strings, podem determinar o seu final ou terem uma interpretação diferente da desejada. Esses caracteres, para que possam aparecer na sua forma original, devem ser precedidos de uma contr-barra. As únicas exceções dentro de um *string* de apóstrofes são o apóstrofe, se ocorrer dentro do *string*, e a contra barra. Para produzir uma contra-barra, deve-se usar duas contra-barras. Para produzir um apóstrofe, deve-se precedê-lo por uma contra-barra.

Strings entre aspas (duplas) não são neutros. Variáveis escalares neles contidos, cifrões, arrobas, porcentagens podem ter uma interpretação diferente se não forem precedidos de contra-barra. Além disso, há várias seqüências de escape formadas por certos caracteres especiais precedidos pela contra-barra para produzir efeitos especiais. Veja na tabela abaixo.

SEQÜÊNCIA	VALOR DO CARACTERE
`\b`	Retrocesso (backspace)
`\t`	Tabulação
`\n`	Nova Linha (new line)
`\f`	Alimentação de Formulário (form feed)
`\r`	Retorno de Carro (carriage return)
`\v`	Tabulação vertical
`\a`	Bell
`\e`	Escape
`\cX`	Qualquer caractere de controle (aqui, `^X`)
`\l`	Próximo caractere em caixa baixa
`\L`	Próximos caracteres em caixa baixa até o `\E`
`\u`	Próximo caractere em caixa alta
`\U`	Próximos caracteres em caixa alta até o `\E`
`\E`	Terminador de `\U` ou `\L`
`\"`	Aspas
`\'`	Aspa (apóstrofe)
`\\`	Contra Barra
`\`*nnn*	O caractere correspondente ao valor octal *nnn*, onde *nnn* é um valor entre 000 e 037.
`\x`*nn*	O caractere de 8-bits *nn*, onde *nn* é de um a dois dígitos hexadecimais.

A terceira categoria de *strings* em *Perl* não representa cadeias de caracteres mas, comandos do sistema. Comandos entre crases `` `dir c:` `` são executados e seu resultado é retornado. Tais escapes tornam o programa dependente de plataforma e devem ser evitados caso se pretenda rodar o programa em mais de uma plataforma.

Comandos do sistema também podem ser executados através do operador `system`:

```
system "dir c:";
```

1.3. Operadores

Perl oferece operadores diferentes para *strings* e números. Os operadores aritméticos são os mesmos usados em *Java*, *C*, *C++* e linguagens semelhantes. As regras de precedência das operações básicas também são as mesmas, mas *Perl* tem regras de precedência próprias para diversos outros operadores.

OPERADOR	FUNÇÃO	OPERADOR	FUNÇÃO
+	Adição	~	Complemento
-	Subtração	=	Atribuição simples
*	Multiplicação	+=	Atribuição com soma
/	Divisão	-=	Atribuição com subtração
%	Resto	*=	Atribuição com multiplicação
**	Exponenciação	/=	Atribuição com divisão
++	Incremento	%=	Atribuição com resto
--	Decremento	&=	Atribuição com AND
!	NÃO lógico	|=	Atribuição com OR
&&	E lógico	^=	Atribuição com XOR
||	OU lógico	?:	Operador ternário *if/then/else*
&	AND	x	Repetição de *string*
^	XOR	.	Concatenação de *string*
|	OR	-*op*	Teste de arquivo (*op:* e, d, ...)
>>	Desloc. de bits à direita	=~	Expr. regular combina
<<	Desloc. de bits à esquerda	=!	Expr. regular não combina

A tabela acima não inclui os operadores usados para realizar comparações booleanas. Para comparar números ou *strings*, os operadores não são os mesmos. A tabela abaixo mostra as diferenças:

COMPARAÇÃO	STRING	NÚMERO
Maior	gt	>
Menor	lt	<
Maior ou igual	ge	>=
Menor ou igual	le	<=
Igual	eq	==
Diferente	ne	!=

Por exemplo:

```
if ($metodo eq "POST") {
    if ($numero >= 25) {
        ....
```

Strings podem ser concatenados usando o operador "." (ponto). Quaisquer números que estiverem envolvidos serão convertidos em *strings* antes da concatenação. Se *strings*, que tiverem partes numéricas, forem usados em operações aritméticas, eles também serão convertidos em números (se for possível, ou seja, se alguma parte desses *strings* representar um número). As partes não numéricas serão descartadas.

O operador x é usado para repetir *strings*. É um operador não comutativo (como o operador de concatenação) e transformará em *strings* qualquer coisa que estiver à sua esquerda. À direita, este operador recebe um número, que informa quantas vezes um determinado *string* deve ser repetido:

```
"Au" x 9; # imprime AuAuAuAuAuAuAuAuAu
"dinheiro" x 3; # imprime dinheirodinheirodinheiro
```

1.4. Variáveis escalares

Variáveis escalares são representadas por um identificador consistindo de um cifrão ($), seguido por uma letra, e depois, talvez, por números e/ou letras. Cria-se uma variável, atribuindo-lhe um valor escalar (número ou *string*). Não é necessário declarar variáveis, basta usá-las. As variáveis não têm tipo. Podem armazenar valores de tipos diferentes a qualquer momento.

```
$a = 13;
$b = $a + 26;
$b++;    # operação de incremento
```

A operação mais comum sobre variáveis escalares é a atribuição. O operador de atribuição em *Perl* é o sinal de igualdade (=). Uma atribuição pode ser combinada com um operador e formar uma atribuição binária:

```
$a = $a + 1; # esta operação, faz a mesma coisa que...
$a += 1; #atribuição binária (atribuição com soma) ... e
$a++;    # incremento unário
```

O operador `chop` pode ser usado apenas com variáveis escalares para arrancar o último caractere de um *string*. Isto é útil quando se lê dados da entrada padrão. A operação atua sobre a variável e retorna o caractere arrancado:

```
$gelado = "abcd";
chop $gelado;       # $gelado agora contém "abc"
$y = chop $gelado;  # $gelado contém "ab" e $y contém "c"
```

Variáveis que ocorrem dentro de *strings* de aspas duplas não são ignoradas. Dentro de tais *strings*, portanto, se houver necessidade de se imprimir um cifrão, é preciso precedê-lo por uma contra-barra. Outra solução é colocar o cifrão dentro de um *string* neutro (de apóstrofes):

```
$a = "texto";
$b = "Eis uma linha de $a"; # produz "Eis uma linha de texto"
$c1 = "Eis uma linha de \$a";  # produz "Eis uma linha de $a"
$c2 = "Eis uma linha de ".'$a';  # produz "Eis uma linha de $a"
```

Variáveis usadas antes de terem um valor atribuído a elas possuem o valor `undef`. Esse valor se traduz como false, zero ou *string* nulo dependendo do contexto onde é utilizado.

Strings de múltiplas linhas podem ser atribuídos a variáveis de duas formas. A primeira, como já vimos, usando atribuição simples:

```
$texto = "
Texto de mais de uma linha.
Segunda linha.
Terceira.
";
```

Uma outra forma é usar o operador "<<", que permite abrir um espaço para um longo *string*, que só termina quando um delimitador escolhido for encontrado sozinho em uma linha. O efeito do bloco abaixo é o mesmo que o anterior:

```
$texto = <<FIM;
Texto de mais de uma linha.
Segunda linha.
Terceira.
FIM
```

Não pode haver qualquer outra coisa na linha que possui o delimitador ou o programa não o encontrará (nem ponto-e-virgula). Deve haver uma quebra de linha imediatamente após o nome e não deve haver qualquer espaço antes.

1.5. Vetores indexados

Um vetor indexado é uma lista ordenada de valores escalares. Cada elemento é uma variável escalar separada com um valor independente, que pode ser de qualquer tipo. Um vetor pode ter qualquer quantidade de elementos. Não é preciso declará-los. Podem crescer ou diminuir à vontade, conforme necessário.

Vetores são representados por literais que consistem de um par de parênteses (o vetor vazio), um par de parênteses contendo um escalar (vetor unitário) ou um par de parênteses contendo vários escalares, separados por vírgulas. Os valores podem ser constantes, variáveis, expressões ou até outros vetores (cujos elementos são incorporados e fazem crescer o vetor):

```
(1, 2, 3, 4, 5)
($a, $b, $c) = (1, 2, 3) # atribui 1 a $a, 2 a $b, etc.
($a + 15, "anta", -17, $x)
(1..4, 10..14) # mesmo que (1, 2, 3, 4, 10, 11, 12, 13, 14)
```

1.6. Variáveis de vetores

Uma variável de vetor contém um único vetor (com sua coleção de valores escalares). Os identificadores são idênticos aos usados com escalares, exceto que, estes, começam com `@` em vez de `$`. Por exemplo:

```
@vetor = ($a, $b, $c, $d);
($x, $y) = @vetor; # $x contém $a e $y contém $b
@maior = (1, @vetor, 2, 3) #  mesmo que (1, $a, $b, $c, $d, 2, 3)
@dois = (1, 2);
($u, $d, %t) = @dois; # contém (1, 2, undef)
```

Uma variável de vetor que ainda não tenha sido inicializada com uma atribuição contém o vetor vazio (sem elementos) representado pelo par de parênteses `()`. Observe que os elementos dos vetores inseridos têm o mesmo nível dos elementos originais. Se o vetor só contém referências escalares (como o segundo e últimos exemplos acima), pode ser usado do lado esquerdo da atribuição. Se um vetor for atribuído a um escalar, este guardará o seu comprimento (número de elementos):

```
$numElem = (1, 20, 46, 89) # $numElem contém 4
```

Os elementos dos vetores são acessíveis individualmente através de seu índice usando colchetes. A referência é o próprio identificador do vetor na sintaxe de escalar (precedido de `$`) e seguido de um par de colchetes `[` e `]` contendo um número (ou expressão que resulta em número) correspondente ao elemento. A contagem dos elementos começa em zero:

```
@planetas = ("mercúrio","vênus","terra","marte");
$x = $planetas[2];         # $x contém "terra"
$planetas[1] = "Ishtar";   # troca vênus por Ishtar

$planetas[5] = "Saturno"; # @planetas agora contém ("mercúrio",
                          # "Ishtar","terra","marte",undef,"saturno")
$a = 2;
$b = $planetas[$a - 1]     # contém "Ishtar"
```

Observe que a variável que tem acesso ao elemento do vetor é escalar. Pode-se usar um variável de vetor, porém, para obter apenas uma parte do vetor original:

```
@dois = @planetas[1, 2]; # @dois contém ($planetas[1], $planetas[2])
@tres = (5,10,15,20,25,30,35,40) [4,5,6]; # @tres contém (25, 30, 35)
```

1.7. Operadores de vetores

O vetor pode ser manipulado como uma pilha usando os operadores `push` e `pop` para acrescentar ou remover os últimos elementos de uma lista:

```
push (@planetas, $novoAstro) ; # empurra $novoAstro no fim de @planetas
$ultimo = pop (@planetas);     # remove o último elemento
push (@x, 1, 2, 3);            # mesmo que (@x, 1, 2, 3)
```

Os operadores `push` e `pop` operam no fim do vetor. Para operar no início, pode-se usar `shift` e `unshift`. O operador `reverse` retorna os elementos do vetor na ordem inversa e `sort` os ordena de acordo com a ordem ascendente dos caracteres ASCII:

```
@z = (1, 2, 3, 10, 20, 30, 100, 200, 300);
@a = reverse @z;   # @a contém (300,200,100,30,20,10,3,2,1)
@s = sort @z;      # @s contém (1, 10, 100, 2, 20, 200, 3, 30, 300)
```

Para que `sort` faça a ordenação de acordo com uma outra regra (numérica, por exemplo), é preciso passar dois argumentos para o operador. O primeiro deve ser uma subrotina (ou corpo da subrotina) que retorne `-1`, `0` ou `1` se o primeiro elemento é menor, igual ou maior, respectivamente, que o segundo.

```
@s = sort &ord_numerica @z;  # &ord_numerica é uma chamada de
                             # subrotina definida localmente
```

Para ordenar pela ordem numérica, pode-se usar o operador auxiliar <=>, no corpo de uma subrotina definida após o `sort` da forma:

```
@s = sort {$a <=> $b} @z;  # (1, 2, 3, 10, 20, 30, 100, 200, 300)
```

Se o operador `chop` for aplicado em um vetor, arrancará o último caractere de cada um de seus elementos.

Expressões pode ser calculadas no contexto escalar ou no contexto de vetor. Por exemplo, se `@x` for impresso, pode-se desejar imprimir seu conteúdo (contexto de vetor) ou seu número de elementos (contexto escalar):

```
@x = ("uga","uga","uh");
print ("Ele disse", @x,".");    # imprime "Ele disse ugaugauh."
print ("Ele disse", "".@x,"."); # imprime "Ele disse 3."
```

No exemplo acima, `@x` foi concatenada com o *string* nulo `""`, o que a transformou em escalar (contém agora o número de elementos do vetor).

Se uma variável de vetor aparecer dentro de uma *string*, será interpretada (assim como ocorre com os escalares). É preciso, portanto, preceder os `@`'s e possíveis `[]`'s com contra-barras (ou colocá-los em *strings* neutros) para que não sejam interpretados como vetores ou elementos de vetor, caso esta seja a intenção:

```
@argos = ("pa","ca"); $argos = "jacaré";
$x = "joao@argos.net";          # imprime "joaopaca.net"
$x = "joao\@argos.net";         # imprime "joao@argos.net"
$y = "A variável é $argos[1]";  #imprime "A variável é ca"
$y = "A variável é $argos\[1]"; #imprime "A variável é jacaré[1]"
```

1.8. Estruturas de controle

As estruturas de controle em *Perl* operam sobre blocos de declarações. Um bloco de declarações é uma instrução composta por várias instruções simples, contidas entre chaves. Um bloco é aceito no lugar de qualquer instrução simples e as suas variáveis (mesmo as criadas dentro do bloco) são globais, a não ser que sejam declaradas locais.

As estruturas de *Perl* são `if-elsif-else`, `unless`, `while`, `until`, `for` e `foreach`. A expressão `if` opera sobre um bloco (que a segue) somente se uma determinada condição (expressa, entre parênteses, após o `if`) for verdadeira. Após o bloco `if` podem existir zero ou mais blocos `elsif`, com outras condições, e, no final, um bloco `else`, complementando os resultados possíveis:

```
if ($x > 10) {
   $b = "é maior";
} elsif ($x < 0) {
   $b = "é menor";
} else {
   $b = "está OK";
}
```

Os valores `undef`, `""` e zero são considerados valores falsos. *Strings* contendo qualquer coisa são considerados verdadeiros, assim como qualquer número diferente de zero.

A expressão `unless` ("a não ser que") é o contrário do `if`. Pode ser usada quando se deseja apenas a parte `else` do `if` (mas ela também pode ter cláusula else):

```
unless ($salario <= 50000) {
 # faça algo somente se $salario não for <= 50000
} else {
 # caso contrário... se salario for <= 50000
}
```

A expressão `while` ("enquanto") permite que se implemente repetições. A expressão `until` ("até que") é o seu oposto. O par `do...while` ("faça ... enquanto") permite que antes do teste, o bloco de instruções seja executado pelo menos uma vez:

```
while($dias < 31) {
   $dias++;
}

until ($dias >= 31) {
   $dias++;
}

do {
   $dias++;
} while($dias < 30);
```

Para repetições em que se sabe previamente o número de vezes em que a operação irá ocorrer, pose-se usar o `for`, que tem a mesma sintaxe que em *C* ou *Java*:

```
for ($i = 0; $i < 100; $i = $i + 10) {
    print "$i carneirinhos... \n";
}
```

A iteração `foreach` permite navegar por uma lista de valores e manipulá-los um por vez como escalar. É ideal para manipular vetores.

```
foreach $galinha (@galinheiro) {
    print $galinha + "\n";
}
```

1.9. Outros operadores de controle de fluxo

Os operadores `last`, `next` e `redo` são usados dentro de uma estrutura de repetição para oferecer meios alternativos de controle de fluxo (além do controle baseado em uma condição verdadeira). O operador `last` serve para forçar a saída de uma repetição (`while`, `do-while` ou `for`). Com `next`, é possível pular partes da estrutura e `redo` permite que se inicie uma nova repetição (deixando de fora trechos que seguem a instrução. Veja os exemplos abaixo:

```
while ($tem_emprego) {
    &bate_ponto;
    if ($demissao) {
        &recebe_indenizacao;
        last;  # cai fora do loop
    }
    &trabalha_dia_inteiro;
    $dias++;
}
# last continua aqui... (não volta mais para o loop)

while ($tem_emprego) {
    &bate_ponto;
    if ($dia_de_folga) {
        &recebe_aviso;
        next;  # pula o resto do loop
    }
    &trabalha_dia_inteiro; # pula!
    $dias++;               # pula!
    # next continua aqui... (segue para fazer o teste do while)
}

while ($tem_emprego) {
    # redo continua aqui... (e pula o teste do while!)
    &bate_ponto;
    if ($ponto_errado) {
        redo;  # pula o resto do loop e o teste
    }
    &trabalha_dia_inteiro;
```

```
    $dias++;
}
```

Os operadores `last`, `next` e `redo` permitem sair na estrutura mais interna. Se for necessário o desvio para um outro bloco (mais externo), é preciso rotulá-lo. O rótulo consiste de um nome, seguido por "`:`" (dois-pontos) que precede o `for` ou `while`.

```
EXT: for ($i = 0; $i < 10; $i++) {
   INT: for ($j = 0; $j < 10; $j++) {
      if ($i == $j) {
          last;  # cai fora do loop atual. Mesmo que last INT;
      } elsif ($i < $j) {
          next EXT; # pula esta rodada do loop externo.
      } else {
           print "($i, $j) "; # imprime (sem quebrar a linha)
      }
   }
   print "\n";  # quebra a linha
}
```

O resultado do programa acima está listado abaixo:

```
(1, 0)
(2, 0) (2, 1)
(3, 0) (3, 1) (3, 2)
(4, 0) (4, 1) (4, 2) (4, 3)
(5, 0) (5, 1) (5, 2) (5, 3) (5, 4)
(6, 0) (6, 1) (6, 2) (6, 3) (6, 4) (6, 5)
(7, 0) (7, 1) (7, 2) (7, 3) (7, 4) (7, 5) (7, 6)
(8, 0) (8, 1) (8, 2) (8, 3) (8, 4) (8, 5) (8, 6) (8, 7)
(9, 0) (9, 1) (9, 2) (9, 3) (9, 4) (9, 5) (9, 6) (9, 7) (9, 8)
```

Além das estruturas `if` e `unless`, é possível usar operadores *booleanos* para construir estruturas de decisão. Os operadores são `&&`, `||`. Veja alguns exemplos:

```
if ($x > 0) { print $x; }       # pode ser escrito como:
$x > 0 && print $x;

unless ($y >= 1) { print $y; }  # pode ser escrito como:
$y >= 1 || print $y;
```

Nas expressões `&&`, o primeiro termo é executado. Se retornar *falso*, o segundo é ignorado e a expressão completa retorna *falso*. Se a expressão retornar *verdadeiro*, o valor final da expressão dependerá da execução do segundo termo. Nas expressões `||` se o primeiro termo for *verdadeiro*, o segundo é ignorado e o valor retornado será *verdadeiro*. Se for *falso*, é preciso executar o segundo que dirá o valor da expressão.

O operador ternário `?:` pode ser usado para substituir o `if-else`:

```
if ($x > 0) { $x++; } else { $x--; } # pode ser escrito como:
($x > 0) ? $x++ : $x--;
```

1.10. Vetores associativos

Vetores associativos são coleções de escalares assim como os vetores comuns, mas eles não têm uma ordem definida. Os seus valores são organizados em pares. Um deles é chamado de chave e é utilizado para recuperar o outro valor.

Uma variável de vetor associativo tem um identificador que inicia com um sinal de percentagem `%`. Cada elemento de um vetor associativo é uma variável escalar distinta, acessada por uma outra variável escalar, chamada de chave (*key*). Os elementos são localizados ou criados fornecendo-se a chave entre chaves `{` e `}`:

```
$elem{"Rússia"} = "Moscou";
$elem{"China"} = "Beijing";
$z = $elem{"Índia"}; #retorna o valor armazenado na chave "Índia"
```

Se o objeto não existir, o valor `undef` é retornado.

Perl não possui uma representação uniforme para um vetor associativo (não é possível controlar a ordem dos elementos). Na criação e na impressão, o vetor associativo é representado como uma lista ou vetor, com um número par de elementos. A ordem desse vetor, porém, não é definida (pode não ser sempre a mesma). Embora seja possível copiar um vetor associativo para um vetor indexado, não há como saber qual elemento será colocado em que posição.

```
%pares = ("Rússia", "Moscou", "Índia", "Delhi", "Itália", "Roma");
@vetor = %pares;
```

1.11. Operadores de associações

É possível realizar operações em vetores associativos com os operadores `keys`, `values` e `each`. O operador `keys` retorna uma lista (vetor indexado) das chaves de um vetor associativo. `values` faz o mesmo com os valores. Se não houver elementos no vetor, uma lista vazia `()` será retornada.

```
%cores = ("vermelho","ff0000","azul","0000ff","verde","00ff00");
@chaves = keys %cores;          # ("vermelho", "azul", "verde")
@valores = values %cores;       # ("ff0000", "0000ff", "00ff00")
```

O operador `each` serve para iterar sobre cada um dos elementos de um vetor associativo. É bastante útil quando utilizado junto com um `for` ou um `while`:

```
while (($cor, $codigo) = each %cores) {
     print "<td bgcolor=\#$codigo>$cor</td>";
}
```

Para remover elementos de um vetor associativo, é preciso usar o operador `delete`:

```
delete $cores{'azul'};
```

1.12. Operador eval

O operador `eval` interpreta uma *string* como código *Perl*. É útil para construir instruções e outras operações para posterior execução:

```
$x = "13.2";
$y = "19.6";
```

```
$op = "+";
eval("$x $op $y"); # resolve a expressão 13.2 + 19.6
```

2. Entrada e saída

Nesta seção apresentaremos os mecanismos básicos para entrada e saída de dados em um programa *Perl*, como leitura da entrada padrão, gravação na saída padrão, controle do sistema de arquivos e leitura e gravação em arquivos.

2.1. Entrada e saída padrão

Nos primeiros exemplos apresentados, mostramos exemplos de leitura de uma linha da entrada padrão usando o operador <STDIN>. Passando esse operador para uma variável escalar, obtém-se a próxima linha da entrada ou undef, se não houver mais linhas:

```
$linha = <STDIN>; #lê próxima linha da entrada padrão
```

Se você atribuir <STDIN> a um vetor, todas as linhas restantes serão lidas e armazenadas em um vetor indexado (lista). Cada linha ocupará o espaço de um elemento, inclusive a nova-linha "\n". Para arrancar as novas linhas de todos os elementos do vetor, pode-se usar chop.

```
@linhas = <STDIN>; # lê todas as linhas da entrada padrão
chop @linhas; #arranca últimos letras de todos os elementos de @linhas
```

Uma forma comum de ler todas as linhas da entrada padrão é fazer isto dentro de um bloco de repetição while:

```
while ($_ = <STDIN>) {
    chop $_;
    #fazer alguma coisa com $_;
}
```

Essa operação é tão comum que não é preciso usar a atribuição $_ = <STDIN>. Sempre que um *loop* contém apenas um operador de entrada de dados <...>, o seu conteúdo é armazenado na variável especial $_. O bloco acima, portanto, também pode ser escrito da seguinte maneira:

```
while (<STDIN>) {
    chop $_;
    #fazer alguma coisa com $_;
}
```

Para ler o conteúdo de arquivos passados na linha de comando, preenchidas no momento da execução do programa, utiliza-se o operador <>. Por exemplo, suponha o seguinte programa type.pl que é chamado da forma:

```
perl type.pl inuteis1.txt inuteis2.txt
```

Se você fizer:

```
while(<>) {
    print $_;
}
```

todo o conteúdo dos dois arquivos será impresso na tela.

Os dados passados na entrada padrão também podem ser obtidos através do vetor `@ARGV`. Esse vetor contém todos os nomes que aparecem após o nome do programa que podem ser obtidos através de `$ARGV[0]`, `$ARGV[1]`, etc. A variável `$ARGV` (sem índice) contém o nome do programa.

Para ler um determinado número de caracteres da entrada padrão (ou de qualquer outro dispositivo) pode-se usar o operador `read`. O nome `STDIN` na verdade é apenas uma constante que serve de meio de acesso à entrada padrão. `read` recebe um vetor que deve conter a origem, o destino e o número de caracteres que devem ser lidos, nessa ordem. Para ler 15 caracteres da entrada padrão para dentro da variável `$cgc`, pode-se fazer:

```
read(STDIN, $cgc, 25);
```

Para imprimir na saída padrão pode-se usar os operadores `print` ou `printf`. O operador `print` pode receber um escalar ou um vetor. Os escalares passados no vetor são concatenados antes de impressos. Os resultados das três instruções abaixo é o mesmo:

```
print "Hello, world!"; # escalar
print ("Hello, world!"); # vetor de um elemento
print ("Hello", "world", "!"); #vetor de três elementos
```

Para imprimir texto formatado, *Perl* dispõe do operador `printf` que funciona da mesma maneira que o `printf` do *C*. A sintaxe inclui valores especiais como "`s`" para *strings*, "`d`" para decimais e "`f`" para ponto-flutuante. Suponha que você tenha as seguintes variáveis:

```
$nome = "Aristóteles";
$idade = 2383;
$pi = 3.14159265359;
```

Usando o operador `printf` da forma:

```
printf("%25s %5d anos. Número: %5.4f\n", $nome, $idade, $pi);
```

obtém-se a seguinte linha formatada (veja tamanhos dos campos em baixo):

```
                Aristóteles  2383 anos. Número:     3.1416
-------------------------------------------------------------
1234567890123456789012345 12345                12345.1234
       25 caracteres      5 dígitos      5 dígitos . 5 decimais
```

O sinal de `%`, se usado dentro do *string* de `printf`, deve vir precedido de `\`, caso contrário, será utilizado para se referir à quantidade de caracteres de uma *string* (sufixo `s`) ou número de dígitos de um decimal (sufixo `d`). Os números de ponto-flutuante podem ser arredondados pelo número de casas especificada após o ponto no campo `%` com sufixo `f`.

A função `printf` permite *imprimir* dados formatados. Em vez de impressos, os dados podem ser atribuídos a uma variável usando `sprintf`. Veja um exemplo:

```
$tf = sprintf("%25s %5d anos. Número: %5.4f\n", $nome, $idade, $pi);
# coloca os dados formatados dentro da variável $tf
print $tf; # imprime os dados formatados (como printf)
```

2.2. Arquivos

Assim como STDIN fornece um meio de acesso à entrada padrão, pode-se criar em *Perl* outras constantes para ter acesso a arquivos. São chamados de *file handles* (alças de arquivo). Além de STDIN a linguagem fornece ainda STDOUT – a saída padrão, e STDERR – a saída padrão de erro. A operação print imprime por *default* na saída padrão. Para imprimir em outro dispositivo, basta informá-lo após o operador:

```
print STDERR "Erro: tente outra vez! \n";
print "Saída normal\n"; # mesmo que print STDOUT "Saída normal\n";
```

O dispositivo de padrão de saída também pode ser alterado usando o operador select:

```
select STDERR; # seleciona STDERR como nova saída padrão
print "Erro: tente outra vez!\n"; # mesmo que print STDERR "...";
select STDOUT; # default agora volta a ser STDOUT
```

Constantes de acesso a arquivos podem ser definidas pelo programador e utilizam-se de um identificador que não possui prefixo (como $, %, @ ou &). Para evitar conflitos com palavras reservadas do *Perl*, recomenda-se usar somente maiúsculas para definir tais constantes.

Para criar novas constantes de acesso a arquivos utiliza-se o operador open. A sintaxe é:

```
open(NOME_DA_CONSTANTE, "nome_do_arquivo_ou_dispositivo");
```

O nome do arquivo pode ser o nome de um dispositivo como um programa que oferece o serviço de e-mail. Depois de criado a constante, ela pode ser usada para se ler o arquivo. Não interessa o conteúdo. Programas em *Perl* podem ler tanto arquivos de texto (7-bit) como arquivos binários (8-bits). Para abrir o arquivo nomes.txt e imprimir todo o seu conteúdo, pode-se fazer:

```
open(ARQNOMES, "nomes.txt");
while (<ARQNOMES>) {     # lê uma linha do arquivo e coloca em $_
    print $_;
}
```

O processo de abertura do arquivo pode falhar. Se isto ocorrer, open retorna undef. Uma forma de evitar um erro caso o arquivo não seja aberto é colocar o open dentro de um if ou unless e só tentar usar a constante criada se open tiver sucesso.

```
if (open(ARQNOMES, "nomes.txt")) {
     ...
} else {
    print "Erro: não foi possível abrir nomes.txt";
}
```

Geralmente, quando isto ocorre, é preciso terminar o programa. *Perl* facilita o processo com o operador die, que, como print, imprime um *string* só que na saída padrão de erro (STDERR) e depois sai do programa sinalizando com um código de erro.

```
unless (open(ARQNOMES, "nomes.txt")) {
   die "Erro: não foi possível abrir nomes.txt";
}
```

Isto também pode ser feito com o operador ||, da forma mais compacta ainda *open or die* (ou abre ou morre):

```
open(ARQNOMES, "nomes.txt") || die "Erro: não pude abrir nomes.txt";
```

Para abrir um arquivo para gravação é preciso indicar a operação adicionando um prefixo ao nome do arquivo. A gravação pode ser para sobrepor (substituir) os dados do arquivo (se ele existir) ou para acrescentar, preservando o que já existe. Em ambos os casos, se o arquivo não existir, ele será criado. O símbolo ">" deve ser usado para indicar gravação com sobreposição e ">>" para indicar gravação sem sobreposição. Para imprimir no arquivo, usa-se `print` com dois argumentos, sendo o primeiro o nome da constante de acesso ao arquivo:

```
open(ARQNOMES, ">>nomes.txt") || die "Erro: não pude abrir
                                      nomes.txt para gravação";
print ARQNOMES ("Bento Carneiro;", "Rua dos Vampiros Brasileiros, 27;");
print ARQNOMES "\n";
print "Nome cadastrado no banco de dados"; # imprime na saída padrão
```

O trecho de código abaixo lê uma linha de cada vez do arquivo `fonte.txt` e copia a linha ao arquivo `destino.txt`.

```
open(FONTE, ">fonte.txt") || die "Erro: não pude abrir fonte.txt";
open(DESTINO, ">destino.txt") || die "Erro: não pude abrir
                                      destino.txt para gravação";
while (<FONTE>) {
    print DESTINO $_;
}
```

Para fechar um arquivo, usa-se o operador `close`. Em geral, o sistema se encarrega de fechá-lo quando sai do *loop*, mas, caso seja necessário realizar o fechamento antes, use:

```
close FONTE;
close DESTINO;
```

É possível realizar diversos testes sobre os arquivos para obter informações sobre eles usando operadores *-op*, onde *op* pode ser uma entre mais de 20 letras disponíveis para testas se o arquivo é executável, se ele pode ser lido, se é uma conexão de rede, se é texto, se é diretório, sua idade, etc. A sintaxe típica é:

```
if (-op $arquivo) {
    # operações se -op for verdadeiro ....
}
```

Por exemplo `-e` serve para testar se um arquivo existe e `-d` retorna um valor verdadeiro se o arquivo for um diretório, então:

```
$arquivo = "papainoel";
if (-e $arquivo && !(-d $arquivo)) {
    # operações se -op for verdadeiro ....
}
```

só será executado se o `papainoel` existir e *não* for um diretório.

A tabela abaixo relaciona os principais operadores de teste de arquivos de *Perl* e seus significados. A maioria retorna uma condição verdadeira ou falsa. Outros retornam um valor numérico.

TESTE	SIGNIFICADO
`-r`	Arquivo ou diretório é acessível (pode ser lido)
`-w`	Arquivo ou diretório pode ser alterado
`-x`	Arquivo ou diretório é executável

TESTE	SIGNIFICADO
-e	Arquivo ou diretório existe
-z	Arquivo existe e tem tamanho zero
-s	Arquivo ou diretório existe e tem tamanho diferente de zero
-f	Operando é um arquivo comum
-d	Operando é um diretório
-l	Operando é um vínculo simbólico
-S	Operando é um soquete de rede
-c	Operando é um dispositivo especial
-T	Arquivo é ASCII
-B	Arquivo é BINARY
-M	Idade do arquivo em dias
-A	Último acesso do arquivo em dias
-C	Última modificação do arquivo em dias

Para mover (mudar o nome) e remover um arquivo ou diretório, *Perl* oferece dois operadores: `unlink` – remove um arquivo, e `rename` – altera o nome de um arquivo.

```
print "Digite o nome do arquivo a ser removido: ";
chop($arquivo = <STDIN>);
unlink ($arquivo);

unlink ("senhas.txt","jegue.gif");
unlink (<*.bak>); #isto é um 'glob'. Veja próxima seção

rename ("despesas.txt", "receita.txt") || die "Isto é ilegal!";
```

2.3. Diretórios

Varios operadores em *Perl* servem para acesso a diretórios. O mais simples é o `chdir`, que permite mudar de um diretório para outro, e navegar no sistema de arquivos:

```
chdir ("/docs/lixo") || die "Não é possível mudar para o lixo!"
```

O comando é dependente de plataforma. Para rodá-lo em *Windows*, é preciso usar a barra invertida a não ser que o programa seja criado para operar como CGI em um servidor *Apache* que, mesmo em *Windows*, interpreta a barra como separador de diretórios. Para criar e remover diretórios, *Perl* oferece respectivamente os operadores `mkdir` e `rmdir`. Na criação de diretórios usando `mkdir`, deve-se informar o formato de permissões *chmod* do *Unix*. Só é possível a remoção de diretórios vazios:

```
mkdir ("zona", 0777); #cria um diretório onde todos têm acesso total
mkdir ("casa", 0755); #cria um diretório onde o dono tem acesso total
                        mas o grupo e outros só podem ler e executar
rmdir("encosto") || die "Não é possível se livrar do encosto!";
```

O recurso do *Unix* de permitir a filtragem de listas de arquivos através do *Shell* é chamada de *globbing*. Isto pode ser usado em *Perl* colocando o padrão de busca entre sinais < e >:

```
@lixo = </docs/*.bak>;
```

Da forma acima, o *glob* retorna uma lista (vetor) contendo todos os nomes que coincidem com o padrão de busca. No contexto escalar (se o *glob* for atribuído a uma variável escalar), o próximo nome de arquivo será recebido.

Para ler o conteúdo de um diretório é preciso abri-lo e para isto, assim como fizemos com os arquivos, precisamos definir uma constante (*file handle*) que permita o acesso ao diretório. O operador `open` não deve ser usado neste caso mas *Perl* oferece um operador chamado `opendir` que tem exatamente a função de abrir diretórios e retornar uma constante que permita manipulá-lo. Uma vez aberto, o conteúdo do diretório pode ser listado com `readdir`.

```
opendir (CAIXA, "/pandora") || die "Você não pode abrir esta caixa!;
while ($males = readdir(CAIXA)) {
   print "$males\n";
}
```

Para sair do diretório, usa-se `closedir` (que geralmente não é necessário, pois o sistema fecha os arquivos e diretórios quando sai do *loop*.

```
closedir(CAIXA);
```

3. Expressões regulares

Expressões regulares representam padrões que devem ser comparados a uma cadeia de caracteres. O resultado da comparação pode ser utilizado simplesmente para tomar decisões (em caso de sucesso ou falha), para obter informações sobre o texto (como, quantas vezes e onde a comparação ocorreu) ou para realizar substituições baseadas no resultado da comparação.

Expressões regulares são comuns em vários programas e ferramentas de sistemas operacionais *Unix* como os *shells* e aplicações como *grep*, *awk*, *emacs* e *vi*. *Perl* possui o conjunto mais completo de expressões regulares entre essas ferramentas *Unix*. Como elas fazem parte da linguagem, também estão presentes nas implementações de *Perl* em *Windows*. Inspiradas em *Perl*, várias outras linguagens modernas, como *JavaScript*, adotam a sintaxe de expressões regulares usadas na linguagem, que hoje também está presente em ferramentas de busca e substituição de processadores de texto, programas de editoração eletrônica e ferramentas de desenvolvimento Web.

3.1. Sintaxe básica

A expressão regular é geralmente representada entre barras "/". Para usá-la, ela deve ser comparada com uma *string* que deve estar armazenada em uma variável escalar (pode ser `$_`) por exemplo:

```
$texto = "Os camelos foram beber água na cachoeira.";
if ($texto =~ /camelo/) {   # expressão regular em negrito!
    print "Existem camelos no texto!";
}
```

O operador =~ define o objeto no qual a expressão regular é calculada. Pode ser qualquer variável ou um dispositivo como `<STDIN>`. Se o objeto e operador =~ é omitido, a operação ocorre sobre a variável global `$_`.

```
$_ = "Os camelos foram beber água na cachoeira.";
if (/camelo/) {
    print "Existem camelos em todo lugar!";
}
```

Geralmente, expressões regulares são mais complexas do que as mostradas acima. Podem conter vários operadores, códigos e caracteres especiais que permitem uma busca mais refinada. Por exemplo:

```
if ($texto =~ /ca.el*o/) {
    print "Existem camelos, cabelos, camellos, cabellllos, ...!";
}
```

A busca com substituição requer expressões mais elaboradas que utilizam os operadores de *transliteração* e de *substituição*. Esses operadores podem ser aplicados para localizar padrões no texto e substituí-los por outros padrões, variáveis, *strings* ou pelo resultado de certas expressões. Contêm várias opções que permitem buscas e

substituições globais, com execução de expressões, etc. Serão apresentados no final desta seção. O exemplo a seguir substitui todas as ocorrências de "cavalo" por "jumento" em `$sitio`:

```
$sitio =~ s/cavalo/jumento/g
```

O sufixo `g` indica que a busca é global.

3.2. Mudança de delimitadores

O delimitador usado por *default* para expressões regulares é a barra "/". Se o texto contido entre barras possui barras, é preciso preceder cada uma delas com uma contra-barra (para anular o efeito da barra). Quando há muitas ocorrências de "/", o código pode ficar ilegível, pois cada "/" se transforma em "\/". É possível, porém, trocar o delimitador por outro como parênteses, chaves, colchetes, cifrões, etc. desde que seja *um* (ou *um par*) de caracteres *não-alfanuméricos*. Para fazer a mudança, o novo delimitador deve vir precedido pelo caractere `m`. O símbolo que seguir o `m` será o novo delimitador.

```
$caminho =~ /\/home\/usuarios\/helder\/public_html\/perl\//; #default: /
$caminho =~ m@/home/usuarios/helder/public_html/perl/@; # delim: @
$caminho =~ m(/home/usuarios/helder/public_html/perl/); # delim: (
```

Caso sejam usados os caracteres "(", "{" ou "[", o caractere de fechamento será o ")", "}" ou "]" correspondente.

Para substituições, transliterações e outras operações semelhantes (seção seguinte), que usam um operador (`s, tr`) com texto entre *três* delimitadores, o `m` é desnecessário. O delimitador desejado deve ser definido logo após o `s` ou `tr`:

```
$x =~ s/\/home\/usuarios\/helder\//\/~helder\//;   # default
$x =~ s(/home/usuarios/helder/)(/~helder/);      # delim: (
$x =~ s#/home/usuarios/helder/#/~helder/#;       # delim: #
$y =~ tr/abcd/ABCD/;               # delimitador default: /
$y =~ tr#abcd#ABCD#;               # delimitador: #
```

A troca de delimitadores só deve ser realizada nas poucas situações onde, de fato, melhora a legibilidade do código. Evite usar como delimitadores caractere que aparecem dentro da expressão.

3.3. Variáveis

O conteúdo de variáveis pode ser passado para expressões regulares da mesma forma como são interpretados dentro de *strings*. É possível, portanto, construir uma expressão regular a partir de *strings* previamente calculados:

```
$texto = "Os camelos passaram a noite numa festa com as llamas.";
print "Digite uma palavra para procurar no texto: ";
$palavra = <STDIN>;
chop $palavra;
if ($texto =~ /\b$palavra\b/) {
    print "A palavra \"$palavra\" foi encontrada no texto!";
} else {
    print "A palavra \"$palavra\" não foi encontrada no texto!";
}
```

O símbolo `\b` é utilizado para marcar os limites de uma palavra (para evitar que a palavra "aspa", por exemplo, combine com "caspa" ou com "aspargos").

3.4. Padrões simples

A comparação mais simples é aquela que procura combinar uma seqüência exata. Por exemplo, encontrar o caractere "a" em uma *string* ou a palavra "camelo". Um exemplo desse tipo de combinação foi mostrado na seção anterior.

O caractere "`.`" (ponto) é utilizado para representar qualquer caractere, com exceção do caractere de nova-linha ("`\n`"). No exemplo mostrado anteriormente, `/ca.elo/` combina com "camelo", "cabelo", "caeelo", "ca&elo", etc.

Pode-se restringir os caracteres permitidos usando classes de caracteres, representadas por uma seqüência de caracteres permitidos entre colchetes `[` e `]`. Somente um dos caracteres da lista pode estar presente no lugar marcado. Por exemplo, `/ca[bmp]elo/` combina com "camelo", "cabelo" e "capelo", mas não com "caBelo" ou "ca_elo". Para representar uma seqüência de caracteres, pode-se usar o caractere "`-`", por exemplo:

```
/[abcdeABCDE]/   # é a mesma coisa que /[a-eA-E]/
/[0-9]/          # é a mesma coisa que /[0123456789]/
```

Pode-se também inverter a seleção, ou seja, permitir qualquer caractere que não esteja na lista. Para isto, basta colocar um circunflexo "^" antes da seqüência:

```
/[^abcdeABCDE]/ # qualquer caractere menos "a", "b", "c", "d" e "e".
/[^0-9]/        # tudo menos números
```

Para incluir os caracteres "`-`", "^", "`\`" e "`]`" na lista, é preciso precedê-los por uma contra-barra.

Várias classes são predefinidas em *Perl* e suas seqüências podem ser substituídas por caracteres de escape, mostrados na tabela abaixo:

SEQÜÊNCIA	CLASSE EQUIVALENTE
`\d`	`[0-9]`
`\w`	`[a-zA-Z0-9_]`
`\s`	`[ \r\t\n\f]`
`\D`	`[^0-9]`
`\W`	`[^a-zA-Z0-9_]`
`\S`	`[^ \r\t\n\f]`

3.5. Seqüências, multiplicadores e alternação

Os símbolos usados na seção anterior permitem combinar no máximo um caractere. Para fazer combinações mais complexas, é preciso usar outros padrões chamados de seqüências, multiplicadores e alternação. A seqüência já foi mostrada anteriormente. Consiste de qualquer seqüência de caracteres entre barras, como `/camelo/`. Para representar zero ou mais ocorrências de um determinado caractere, utiliza-se o mais simples dos multiplicadores, o asterisco "`*`". Além dele existem o sinal de mais "+" que representa um ou mais (o caractere anterior), e o ponto de interrogação "?", que significa zero ou uma ocorrências do caractere anterior. A tabela abaixo fornece um resumo desses símbolos:

MULTIPLICADOR	MULT. GERAL	RESULTADO
`c*`	`c{0,}`	zero ou mais "c"

MULTIPLICADOR	MULT. GERAL	RESULTADO
`c+`	`c{1,}`	um ou mais "c"
`c?`	`c{0,1}`	zero ou um "c"

Por exemplo, a expressão regular `/ca+me*los?/` combina com "camelo", "camelos", "cameeelos" e "caaamlo", mas não com "cmelo" ou "cameloss".

A substituição usando multiplicadores é sempre realizada com o máximo de elementos possível que combine com o padrão. Por exemplo, a expressão regular de substituição

```
s/a*h/u,
```

quando aplicada à *string*

```
$_ = "Baaaaaaaaah, tchê!";
```

substituirá todos os 10 a's consecutivos com uma única letra "u". Se a intenção é substituir grupos menores, é preciso usar o multiplicador geral, definido entre chaves `{` e `}`. Com ele, pode-se definir uma faixa de valores indicando a quantidade de elementos a serem substituídos. *Exemplos*:

```
s/a{5}h/u    # exatamente 5 "a": Buuh, tchê!
s/a{2,4}h/u  # de 2 a 4 (de preferência) "a": Buuuh, tchê!
s/a{0,3}h/u  # de 0 a 3 "a"
s/a{6,}h/u   # 6 ou mais "a"
```

Para combinar com uma lista de alternativas possíveis de caracteres, pode-se usar as classes de caracteres (entre colchetes). Para combinar entre alternativas de seqüências, porém, as classes não servem. Para isto, usa-se um grupo chamado de alternação, onde as alternativas são separadas pelo caractere "`|`".

```
/dia|tarde|noite|madrugada/ # combina com uma das alternativas
/(segunda|terça|quarta|quinta|sexta)-feira|sábado|domingo/
# parênteses definem precedência. Opções serão segunda-feira,
# terça-feira, ..., domingo.
```

3.6. Âncoras

Há quatro caracteres especiais que ancoram um padrão. Pode-se identificar o início da *string*, o final e a exigência ou não de um limite (início ou final) de uma palavra.

A tabela abaixo relaciona os quatro tipos de padrão.

ÂNCORA	FUNÇÃO
`^`	Início da cadeia de caracteres
`$`	Fim da cadeia de caracteres
`\b`	Fim ou início de palavra
`\B`	Não há fim ou início de palavra

O marcador "`\b`" marca o início ou o final de uma palavra. Já o marcador marca a inexistência desse limite. Veja alguns exemplos:

```
/pão\b/ # combina com pão mas não com pãozinho
/\bpão/ # combina com pão e pãozinho mas não com sapão
```

```
/\bpão\b/ # combina com pão mas não com pãozinho nem sapão
/\bpão\B/ # combina com pãozinho mas não com pão francês
/\Bpão\B/ # combina com sapãozinho mas não com um pão francês
```

Para representar o início de uma *string*, usa-se o circunflexo "^". Este caractere só tem significado especial quando aparece no início da expressão regular. Em qualquer outro lugar, representa o caractere circunflexo. Se for necessário localizar o caractere circunflexo no início do padrão, deve-se precedê-lo por uma contra-barra.

```
/^morcego^/   # combina com morcego^ no início da string
/\^morcego^/  # combina com ^morcego^ em qualquer lugar
```

O cifrão representa o final da cadeia de caracteres. Só faz sentido usá-lo no final de um a expressão (em qualquer outro lugar será interpretado como o prefixo de uma variável escalar).

```
/R$/   # combina com R no fim da string
/R\$/  # combina com R$ em qualquer lugar da string
```

Quando houver vários tipos diferentes de operadores de grupo (multiplicadores, âncoras, alternação, seqüências) sempre pode-se usar parênteses para resolver o problema de operações que devem ser realizadas antes de outras. Há uma ordem de precedência para esses operadores que está mostrada na tabela abaixo:

	OPERADORES	TIPO
1	( )	parênteses
2	+ * ? {m,n}	multiplicadores
3	abcde	seqüências
	^ $ \b \B	âncoras
4	|	alternação

Veja abaixo alguns exemplos de expressões regulares e o efeito dos parênteses e das regras de precedência:

```
/a|b*/        # um único a ou seqüência de bbbbb...
/(a|b)*/      # seqüência de aaa... ou de bbb...
/abc*/        # ab, abc, abcc, abccc, abcccc ...
/(abc)*/      # "", abc, abcabc, abcabcabc, ...
/^x|y/        # x no início da string ou y em qq. lugar
/^(x|y)/      # x ou y no início da string
/a|bc|d/      # a ou bc ou d
/(a|b)(c|d)/ # ac, ad, bc ou bd
```

Importante: Não confunda `/^abc/` com `/[^abc]/`. A primeira expressão combina com a *string* "abc" no início da linha. A segunda combina com *um caractere* apenas que *não* seja "a", "b" ou "c".

3.7. Parênteses como memória

Além de controlar a precedência das operações de grupo, parênteses também são usados como memória para armazenar o resultado de expressões para uso posterior. Para recuperar um padrão previamente armazenado, é preciso utilizar uma contra-barra seguida por um inteiro (`\1`, `\2`, `\3`, etc.) representando a ordem em que a expressão (ou substring) ocorre. Cada par de parênteses utilizado, mesmo se utilizado apenas para determinar precedência, é lembrado e recuperado por um número inteiro. Veja alguns exemplos

```
/<(b)>.*<\/\1>/ # combina com qualquer coisa entre <b> e </b>
/25(.)12\1999 13(:)15\259/ # 25-12-1999 13:15:59 ou 25/12/1999 ...
/C(.*)s e D\1s/ # Cobras e Sobras, Cucos e Sucos, Cantos e Santos
```

As principais aplicações dos parênteses como memória ocorrem em substituições:

```
$pagina = "<html><head> ... </body></html>";
$pagina =~ s/<([^>]*.)>/&lt;\1&gt;/g
# troca todos os <...> ('...' ñ contém ">") por &lt;...&gt;
```

Depois de uma combinação bem sucedida (como o exemplo acima), as variáveis somente-leitura `$1`, `$2`, `$3`, etc. são definidas com os mesmos valores de `\1`, `\2`, `\3`, etc. Elas podem então ser usadas em outras partes do programa.

```
$bichos = "As cobras e os ratos estão com sede.";
$bichos =~ /.s c(.*)s e .s r(.*)s /; # observe o espaço após o "s"
print "Os g$2s devoram as s$1s do restaurante\n";
```

As variáveis também podem ser lidas em um contexto de vetor:

```
$_ = "As cobras e os ratos estão com sede.";
($primeiro, $segundo) = /.s c(.*)s e .s r(.*)s /;
# ou @valores = /.s c(.*)s e .s r(.*)s /;
```

No exemplo acima, `$primeiro` contém o valor armazenado em `$1` e `$segundo` contém o valor em `$2`. Há ainda outras três variáveis predefinidas que obtém informações adicionais sobre a expressão regular. Todas estão mostradas na tabela abaixo.

VARIÁVEL	FUNÇÃO
`$1, $2, $3, ...`	Guardam os valores armazenados nos parênteses
`$&`	Guarda o resultado da expressão regular
`$``	Guarda o texto antes do resultado (crase)
`$'`	Guarda o texto após o resultado (apóstrofe)

As variáveis são definidas após cada expressão calculada. Se for necessário utilizá-las mais adiante no programa, deverão ser copiadas para outras variáveis. Veja um exemplo:

```
$texto = "Os camelos passaram a noite numa festa com as llamas";
$texto =~ /ca(.el)*o/;
print "Primeiro texto armazenado: $1\n"; # imprime "mel"
print "Texto localizado: $&\n"; # imprime "camelos"
print "Texto anterior: $'\n";  # imprime "Os "
print "Texto posterior: $`\n";  # imprime " passaram ... llamas"
```

3.8. Outros operadores

O sufixo `i` (de *ignore case*) pode ser acrescentado após a última barra de uma expressão regular para torná-la insensível ao formato maiúsculo ou minúsculo dos caracteres (*case-insensitive*). Veja um exemplo:

```
print "Digite o que você quer fazer: ";
$nome = chop <STDIN>;
if ($nome =~ /^listar/i) {
```

```
        # Usuário pode digitar "Listar" ou "LISTAR RESULTADOS" ou ...
    }
```

3.9. Substituições

O operador de substituição, representado pelo prefixo `s`, colocado antes de uma expressão regular de duas partes, tem a seguinte sintaxe básica:

```
s/texto_antigo/texto_novo/
```

A expressão acima altera apenas a primeira ocorrência de `texto_antigo`. Para localizar e substituir todas as ocorrências, utiliza-se o sufixo `g`:

```
s/texto_antigo/texto_novo/g
```

O sufixo e é usado para tratar a segunda parte de um substituição como uma expressão. Na substituição, a segunda parte é executada:

```
s/(camelos)/chop $1/e   # substitui camelo por camelos
```

A tabela abaixo contém vários sufixos freqüentemente usados nos operadores de substituição. Os sufixos podem ser colocados lado a lado em qualquer ordem. O resultado de qualquer uma das duas expressões abaixo é o mesmo:

```
s/texto_antigo/chop($texto). !/ge;
s/texto_antigo/chop($texto). !/eg;
```

SUFIXO	FUNÇÃO
`i`	Ignora formato caixa-alta ou caixa-baixa em qualquer expressão regular.
`e`	Trata a segunda parte da substituição como expressão *Perl* (e executa).
`g`	Realiza a substituição em todo o *string* (busca e substituição global)

3.10. Transliteração

O operador de transliteração é útil para realizar substituições simultâneas entre caracteres. Para realizar uma transliteração, usa-se o comando `tr`, seguido de dois argumentos entre barras (ou outro delimitador qualquer), assim como o comando `s`. Os operandos consistem de uma seqüência antiga e uma nova. O primeiro caractere da nova seqüência substituirá todas as ocorrências do primeiro caractere da antiga seqüência encontrados no texto-objeto da substituição (ou a variável `$_`). O mesmo ocorrerá com os segundos caracteres, os terceiros, etc.

```
$texto = "O rei roeu os rabos dos ratos de Roma.";
$texto =~ tr/aeiou/AEIOU/;
# $texto agora contém "O rEI rOEU Os rAbOs dOs rAtOs dE ROmA."
```

É possível especificar uma seqüência ordenada usando o hífen. Por exemplo, tanto faz usar `0123456789` ou `0-9`. Se a primeira seqüência tiver mais caracteres que a segunda, o último caractere da segunda seqüência será repetido para cada caractere da primeira seqüência que não tiver um correspondente na segunda. Veja alguns exemplos:

```
$texto = "O rato roeu a roupa do rei de Roma.";
$texto =~ tr/aeiou/AEIOU/; # 26 cars. para 5
# $texto agora contém "O UAUO UOEU A UOUUA UO UEI UE ROUA."
```

Pode ser usado o valor de retorno de uma transliteração, que contém o número de caracteres afetados. Para obter esse valor, basta atribuir o resultado da expressão à uma variável:

```
$_ = "O rei roeu os rabos dos ratos de Roma.";
$num = tr/aeiou/AEIOU/; # num contém 14 e
# $_ contém "O rEI rOEU Os rAbOs dOs rAtOs dE ROmA."
```

É possível usar o operador `tr` para obter o número de caracteres que combinam com a primeira expressão, sem alterar a *string*, mantendo o segundo argumento da operação vazio:

```
$_ = "O rei roeu os rabos dos ratos de Roma.";
$num = tr/aeiou//; # num contém 14 e $_ não foi alterado.
```

Na tabela abaixo estão as principais opções que podem modificar o operador `tr` e sua função. Assim como as opções de `s`, elas são listadas depois da segunda seqüência, em qualquer ordem:

```
$texto =~ tr/aeiou/AEIOU/cds;
```

SUFIXO	FUNÇÃO
d	Remove caracteres que não aparecem na segunda parte da transliteração.
c	Resultado é conjunto de todos os caracteres que não aparecem na primeira parte da transliteração.
s	Converte múltiplas cópias seguidas de uma letra na segunda parte de uma transliteração como uma só.

A melhor forma de ilustrar o funcionamento de `tr` é através de exemplos. Considere a seguinte *string*:

```
$_ = "O rei roeu os rabos dos ratos de Roma.";
```

A opção `d` elimina quaisquer caracteres listados na primeira seqüência que não têm correspondente na segunda. Abaixo, apenas `aeiou` combina. Sem a opção `d`, a letra `U` substituiria cada letra não encontrada.

```
tr/aeiou/bcdfghjklmnpqrstvwxyz/AEIOU/d; #apaga caracteres
# $_ agora contém "O EI OEU O AO O AO E ROA."
# (os caracteres a-z exceto vogais foram eliminados)
```

A opção `c` considera o complemento (a negação) da primeira seqüência em relação ao universo de caracteres (256) disponíveis.

```
tr/a-z/_/; # sem opções
# $_ agora contém "O ___ ____ __ _____ ___ _____ __ R___."
tr/a-z/_/c; #substitui complemento (caracteres não listados)
# $_ agora contém "__rei_roeu_os_rabos_dos_ratos_de__oma_"
# (todos os caracteres exceto a-z (O,R, ,.) foram convertidos em _
```

A opção `s` ignora repetições sucessivas de uma mesma letra.

```
tr/aeiou/x/;  # sem opções
# $_ agora contém "O rxx rxxx xs rxbxs dxs rxtxs dx Rxmx."
$_ = "O rei roeu os rabos dos ratos de Roma.";
tr/aeiou/x/s; # ignora caracteres repetidos
```

```
# $_ agora contém "O rx rx xs rxbxs dxs rxtxs dx Rxmx."
$_ = "O rei roeu os rabos dos ratos de Roma.";
tr/a-z/x/s; # ignora caracteres repetidos
# $_ agora contém "O x x x x x x x Rx."
```

É possível fazer substituições sofisticadas combinando mais de uma opção:

```
$_ = "um     dois   três  quatro        cinco.";
tr/a-zA-Z\./ /cs; # somente espaços, ignora repetições
# $_ agora contém "um dois tr s quatro cinco."; # ê ñ está entre a-z

$_ = "um     dois   três  quatro        cinco.";
tr/a-zA-Z\.ê/ /cd; # elimina todos os espaços
# $_ agora contém "umdoistrêsquatrocinco.";
```

3.11. Operadores split e join

O operador `split` recebe uma *string* e uma expressão regular. Ele faz uma busca global pela expressão e retorna um vetor contendo as partes da *string* que não combinam com a expressão. O resultado da expressão regular serve, então, como um separador para transformar *strings* em vetores:

```
$registro = "nome; endereço; telefone; email";
@campos = split(/;/, $registro);
($nom, $end, $tel, $ema) = @campos;
```

O valor *default* para o operador `split` é "um ou mais espaços consecutivos", ou seja `/\s+/`. O objeto *default* é a variável `$_`. As duas operações a seguir são, portanto, equivalentes:

```
$_ = "um   dois         três    quatro                  cinco";
@nums = split(/\s+/, $_);  # eh a mesma coisa que...
@nums = split; # resultado: ("um","dois","três","quatro","cinco")
```

O operador `join` é o contrário do `split`. Enquanto aquele separa, este junta. Os argumentos são: um vetor que terá seus elementos concatenados e uma *string*, que será repetida entre os elementos do vetor.

```
$linTab = join("</td><td>", @nums);
# res.: um</td><td>dois</td><td>três</td><td>quatro</td><td>cinco
print "<table border=1><tr><td>".$linTab."</td></tr></table>";
```

4. Subrotinas e bibliotecas

4.1. Subrotinas

O operador `sub` é utilizado em *Perl* para definir um procedimento ou subrotina que pode ser chamado de outra parte do programa ou de outro programa (se incluído em uma biblioteca). Por exemplo:

```
sub soma {
   print 2+2;
}
```

define a subrotina soma. Para chamá-la, utiliza-se um identificador com o nome da subrotina precedido do caractere "`&`".

```
&soma; # imprime 4.
```

Uma subrotina pode ter qualquer número de instruções, simples ou compostas. O último valor resultante de uma expressão é sempre retornado. Por exemplo, na subrotina:

```
sub soma {
   2+2;
}
```

a chamada

```
$res = &soma;
```

armazena o valor 4 em `$res`.

Uma subrotina termina assim que chega ao fim. Para escapar antes, pode-se usar o operador `return` que também pode ser usado para devolver um valor, ao deixar a operação:

```
sub soma {
   return 2+2;
}
```

O valor de retorno não precisa ser um escalar. Pode também ser um vetor.

Algumas subrotinas podem requerer parâmetros passados na sua chamada. Uma rotina de soma mais útil seria chamada da forma:

```
$res = &soma(13,4);
```

O argumento que segue a chamada da função é um vetor. Ele será automaticamente armazenado no vetor global `@_`. Dentro da subrotina, os valores passados podem ser recuperados lendo os valores desse vetor:

```
sub soma {
   ($x, $y) = @_;
   return $x + $y;
}
```

Dentro da subrotina, `@_` é local. As outras variáveis definidas dentro da subrotina, porém, não são. Para declarar variáveis como locais é preciso usar o operador local. A forma típica de construir funções que recebem parâmetros em *Perl*, é, portanto:

```
sub soma {
   local($x, $y, @extras) = @_; #$x e $y são locais (como @_)
   # resto da função.
}
```

A variável `@_` também pode ser referenciada em partes escalares, utilizando os identificadores `$_[0]`, `$_[1]`, etc. O uso de elementos do vetor `@_` que não existem não provoca erros (todos contém `undef`).

O seguinte exemplo mostra uma subrotina que pode aceitar qualquer número de argumentos.

```
sub somatorio {
   local ($sum);
   $sum = 0;
   foreach $_ (@_) {
      $sum += $_;
   }
   return $sum;
}
```

Para chamar a função acima, pode-se usar qualquer vetor ou expressão que resulte em vetor, para passar os parâmetros após o nome da função:

```
&somatorio(4, 8, 10); # soma 4 + 8 + 10
&somatorio(1..20);    # soma 1+2+3+...+19+20
```

4.2. Bibliotecas

Trechos de código em *Perl* definidos em outros arquivos podem ser incluídos em um programa usando a instrução `require`. Esse recurso é extremamente útil para a definição de bibliotecas de subrotinas e variáveis globais que devem ser compartilhadas por mais de um programa.

Qualquer programa em *Perl* pode ser incluído em outro usando `require`. É preciso, porém, que a última linha do código incluído contenha um valor definido e positivo (o valor é retornado pelo `require`). Tipicamente, os arquivos a serem incluídos consistem de uma coleção de subrotinas (blocos `sub { ... }`). Depois das subrotinas, deve haver mais uma linha no fim do arquivo:

```
1;
```

A linha acima é o suficiente para que o arquivo possa ser usado como biblioteca e importado com `require`:

```
require "biblio.pl";
```

Na distribuição original do *Perl* (desde a versão 4) há várias bibliotecas que podem ser importadas usando `require`. Tipicamente, os arquivos têm a extensão `.pl` ou `.ph`. A variável global `$INC[0]` (do vetor `@INC`)

contém o diretório onde tais bibliotecas estão armazenadas (nas distribuições padrão do *Perl* para *Unix* ou no *ActivePerl* para *Windows*).

Várias outras bibliotecas úteis podem ser encontradas na Internet. Uma das mais populares é a biblioteca `cgi-lib.pl`[1], que contém várias subrotinas úteis ao tratamento de dados em programas CGI. Outras como `oraperl.ph` e `sybperl.ph` contém subrotinas e variáveis úteis para o acesso a bancos de dados *Oracle* e *Sybase*, respectivamente. Também podem ser encontradas na Internet[2].

[1] Veja na Internet em http://www.cgi-lib.org. Há também bibliotecas para C e outras linguagens. Para manipulação de "cookies" via CGI, veja a biblioteca Perl cookie.lib em http://www.worldwidemart.com/scripts/

[2] http://src.doc.ic.ac.uk/packages/perl/db/perl4/. Veja também os módulos (orientados a objetos) do Perl 5.

5. Transformação de dados

Uma das principais vantagens de *Perl* como linguagem preferencial para uso na Web (em programas CGI) é a quantidade de recursos que possui para realizar transformação e formatação de dados. Nesta seção serão apresentados os principais operadores para manipulação e formatação de *strings*.

5.1. Manipulação de texto

Três operadores existem para localizar e extrair trechos de um texto mais longo. As funções `index` e `rindex` retornam um número indicando a posição onde tem início uma determinada seqüência de caracteres dentro de um texto maior. A única diferença entre as duas funções é a ordem em que realizam a busca. `index` procura o texto do início até o fim da *string* e `rindex` faz a busca no sentido contrário. A sintaxe geral é:

```
$pos = index($str, $substring); # ou rindex($str, $substring);
```

O número retornado será um valor inteiro maior ou igual a zero e menor que o comprimento total da *string*. Se a *substring* não for encontrada, o valor retornado será `-1`. Se houver mais de uma ocorrência do texto procurado, apenas a posição do primeiro encontrado será retornada (que poderá ser diferente caso seja usado `index` ou `rindex`). É possível iniciar a busca a partir de uma posição específica, informada como terceiro argumento:

```
$pos = index($str, $subs, $ini); # ou rindex($str, $subs, $ini);
```

Veja alguns exemplos usando `index` e `rindex`:

```
$texto = "Abandonai toda esperança, ó vós que entrais!";
$num1 = index($texto, "an");          # $num1 contém 2
$num2 = index($texto, "an", $num+1); # $num2 contém 20
$num3 = rindex($texto, "an");         # $num3 contém 20
$num4 = rindex($texto, "an", $num);  # $num4 contém 2
```

A função `substr` extrai uma parte de um texto maior a partir de suas posições inicial e final. A sintaxe básica é:

```
$novoString = substr($string, $inicio, $fim);
```

As variáveis `$inicio` e `$fim` devem ser inteiros. Se `$início` for menor que zero, a contagem começa a partir do final da *string*. `$fim` deve ser maior que zero para que o resultado retorne uma *string* não-vazia. Veja alguns exemplos:

```
$frag = substr($texto, $num1, $num2); # ou substr($texto, 2, 20);
# $frag contém "andonai toda esper" (caracteres 2 a 19)
```

```
$fim = substr($texto, -8, 6); # contém "entrai"
# iniciando na 8a. letra a partir do fim, retornar as próximas 6
```

O operador `substr` também pode aparecer do lado esquerdo da operação de atribuição. Nessa situação, ele pode receber um novo texto que irá substituir a *substring* no texto original. O texto novo pode ser maior ou menor que o texto substituído:

```
$texto = "Abandonai toda esperança, ó vós que entrais!";
substr($texto, 0, 7) = "Deix"; # 4 letras no lugar de 7
# texto contém "Deixai toda esperança, ó vós que entrais!";
substr($texto, -8, 4) = "cheg";
substr($texto, 0, 4) = "Pass";
substr($texto, 12, 20) = "a grana";
# texto contém "Passai toda a grana, ó vós que chegais!";
```

5.2. Formatos

Um formato é um gabarito (*template*) de impressão. Permite criar um *layout* para posterior formatação de texto. Com formatos pode-se definir como o texto será posicionado na página e fixar previamente títulos de colunas, rótulos cabeçalhos e rodapés.

Antes que um formato possa ser usado ele precisa ser definido. Uma vez definido, pode receber os dados que serão posicionados em seus campos. Finalmente, ele pode ser chamado e impresso.

Um formato é definido através da palavra reservada `format`, seguida de um identificador que será o nome de um formato. Segue a definição do formato (que pode ocupar zero ou mais linhas). A definição termina com um ponto "`.`" isolado no início de uma linha. A definição contém um conjunto de *campos fixos* (linhas de texto), que contém valores fixos que devem ser impressos. Essas linhas podem também conter *campos variáveis* que são reservados através de operadores de formatação. Após uma linha que contém campos variáveis, deve haver outra linha contendo as variáveis que devem substituí-los separadas por vírgulas. Veja um exemplo de formato abaixo:

```
format ETIQUETA =
+-----------------------------------------------+
| Nome: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< |
$nome
| Endereço: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< |
$endereco
| Cidade: @<<<<<<<<<<<<<<< UF: @< CEP: @<<<<-@<<|
$cidade, $uf, $cep1, $cep2
+-----------------------------------------------+
.
```

O operador `@<<<` reserva espaço para um determinado número de caracteres alinhados pela esquerda. `@` representa um caractere, `@<` representa dois, e assim por diante. Se uma linha de campo fixa contiver 3 campos variáveis `@`, ela deve vir seguida de uma linha com 3 variáveis escalares separadas por vírgulas.

Uma vez definido o formato, ele pode ser chamado com o operador `write`. Antes ele deve ser preenchido com os dados que irá formatar. Isto é feito atribuindo valores às variáveis globais utilizadas pelo formato. Os dados podem ser lidos de fontes externas. Por exemplo, suponha um arquivo com vários registros separados por ponto-e-vírgula:

```
Marie Curie;Rua do Césio, 137;Vila Tchernobyl;PE;50213-320
Hans Staden;Rua Potiguares, 13;São Paulo;SP;01234-970
```

Se essas informações estão em um arquivo `dados.txt`, podem ser lidas pelo programa em *Perl* usando `open`. O resultado do formato deve ser redirecionado para um arquivo (ou qualquer dispositivo) que será aberto para gravação. O nome do descritor de arquivo deve ser o mesmo nome do formato:

```
open (ETIQUETA,">etiquetas.txt") || die "Não pude criar formato";
open (DADOS,"dados.txt") || die "Não pude abrir arquivo";
while (<DADOS>) {   # lê uma linha do arquivo para $_
   chop; # remove \n de $_
   ($nome, $endereco, $cidade, $uf, $cep) = split(/;/);
   ($cep1, $cep2) = split(/-/,$cep);
   write ETIQUETA; # envia variáveis lidas para formato
}
```

Depois do código acima, o arquivo `etiquetas.txt` deve conter:

```
+---------------------------------------------+
| Nome: Marie Curie                           |
| Endereço: Rua do Césio, 137                 |
| Cidade: Vila Tchernobyl  UF: PE CEP: 50213-320|
+---------------------------------------------+
+---------------------------------------------+
| Nome: Hans Staden                           |
| Endereço: Rua Potiguares, 13                |
| Cidade: São Paulo        UF: SP CEP: 01234-970|
+---------------------------------------------+
```

Esta seção apenas apresentou os conceitos básicos dos formatos *Perl*. Há vários outros operadores e maneiras diferentes de manipular, construir e utilizar formatos. A tabela abaixo oferece uma lista dos principais operadores para a definição dos campos de um formato.

OPERADOR	FUNÇÃO
`@<<<<`	Reserva espaço para 5 caracteres alinhados pela esquerda.
`@>>>>`	Reserva espaço para 5 caracteres alinhados pela direita.
`@||||`	Reserva espaço para 5 caracteres alinhados pelo centro.
`@####.##`	Reserva espaço para campo numérico com 5 casas antes do ponto decimal e duas casas após o ponto.
`@*`	Reserva espaço para informação que ocupa múltiplas linhas.
`^<<<<,`	Reserva espaço para campos preenchidos (campos multi-linha com larguras definidas).
`^>>>>,`	Cada linha deverá ser seguida por uma linha contendo a variável escalar que contém os
`^||||`	dados (mesmo repetida).
`~`	Suprime uma linha em campos preenchidos caso ela seja vazia.
`~~`	Repete o último formato (campos preenchidos) caso o texto exceda o espaço reservado previamente.
`.`	Termina a definição de um formato (se sozinho no início de uma linha)

6. Módulos

Enquanto *Perl 4* é uma linguagem totalmente orientada a procedimentos, *Perl 5* inclui todo o *Perl 4* e acrescenta uma nova estrutura que permite a construção de programas orientados a objetos.

Objetos permitem uma maior reutilização de código e a realização de tarefas complexas em menos linhas de código e de forma mais simples. Mas a programação orientada a objetos também introduz uma complexidade adicional formada por novos conceitos, termos e técnicas de programação que estão além dos nossos objetivos neste tutorial, já que nosso objetivo é oferecer recursos suficientes ao desenvolvimento de aplicações CGI típicas.

Nesta seção apresentaremos uma visão superficial dos módulos de classes – um recurso orientado a objetos do *Perl 5* – porque são úteis no desenvolvimento de aplicações em *Perl* mais sofisticadas como *gateways* de acesso a bancos de dados ou de geração de imagens, com aplicações na Web através de CGI. Vários módulos também estão disponíveis para a manipulação de dados recebidos por formulários HTML. O objetivo desta seção é, portanto, mostrar como utilizar esses módulos. Não será abordada a construção de classes e pacotes em *Perl*.

6.1. O que são módulos de classe

Módulos de classe são arquivos especiais (chamados de *pacotes*) contendo código em *Perl* e que devem ser usados para o armazenamento de funções e a construção de *objetos*. Um objeto é algo que pode armazenar dados e ocultar operações complexas, fornecendo uma lista finita de operações que podem ser realizadas sobre ele e sobre os dados que contém. As operações de um objeto são chamadas de *métodos*. Os dados que armazena, em geral, só podem ser alterados pelos métodos, que são a sua *interface pública*. Um programa orientado a objetos é, portanto, um algoritmo que, em vez de ser definido em função de uma seqüência de operações, é caracterizado pela interação entre objetos. Tais programas podem apresentar-se bem mais simples que os programas procedurais pois a maior parte da complexidade fica dentro dos objetos que não aparecem no programa principal.

Para utilizar um objeto é preciso primeiro criá-lo. Uma classe é o molde usado para isto. A *classe* contém a definição dos métodos do objeto e uma função especial chamada de *construtor*. O construtor é utilizado para criar novos objetos a partir da classe. Cada objeto tem sua própria cópia dos métodos e variáveis definidos na classe.

Em *Perl*, classes são definidas dentro de módulos. Módulos podem conter pacotes ou subpacotes que, por sua vez, contém as classes. Para usar um módulo é preciso utilizar a instrução use e o nome do(s) pacote(s) definido(s) nele. Por exemplo, para usar o módulo `ODBC.pm` que define o pacote `Win32::ODBC` é preciso importá-lo com:

```
use Win32::ODBC;
```

Os pacotes que se comportam como classes devem conter funções e construtores para a construção de objetos. No caso do módulo acima, o pacote define uma instrução `new` que funciona como construtor (é uma convenção criar construtores com este nome). A instrução requer o parâmetro `Win32::ODBC` (nome da classe – outra convenção) seguido pelo nome de uma fonte de dados ODBC. Para criar um novo objeto com esta classe, portanto, é preciso fazer:

```
$objeto = new Win32::ODBC("nome_DSN");
```

Depois que um objeto é criado, seus métodos podem ser chamados. Diferentemente dos métodos de classe (ou funções) como o construtor `new`, acima, os métodos têm que ser chamados em relação ao objeto ao qual pertencem. O operador utilizado para isto é "->":

```
$objeto->método("arg1", 99, $arg3);
$objeto->metodo2;
```

Os objetos em *Perl* são destruídos automaticamente através de um sistema de coleta de lixo, portanto, o programador não precisa se preocupar em destruir objetos criados.

6.2. Como usar módulos

É preciso conhecer quais os métodos, classes e objetos definidos em um módulo antes de usá-lo. Se o módulo faz parte da distribuição original do *Perl*, deve haver documentação disponível no diretório onde o interpretador foi instalado. Em outros casos, a documentação deve acompanhar a distribuição do módulo.

A instalação de um módulo (que não esteja disponível na distribuição *Perl*) pode consistir da simples cópia de um arquivo com extensão `.pm` para um subdiretório de `[`*dir. instalação*`]/perl/lib/`, de vários outros arquivos ou até da compilação de arquivos. É preciso consultar a documentação de cada módulo.

Na distribuição original do *Perl 5* (5.004) há vários módulos. Muitos fazem a mesma coisa de formas diferentes; uns são mais fáceis de usar, mais completos ou mais eficientes do que outros. Nas subseções a seguir, são apresentados alguns módulos úteis para o desenvolvimento de aplicações CGI.

6.3. Módulos para uso com CGI

Quase todas as distribuições do *Perl 5* (inclusive o *ActivePerl for Windows*) possuem um módulo `CGI::*` ou similar com diversas classes úteis para lidar com dados de formulários e *cookies* (`CGI::Cookie`). Esses módulos podem ser usados no lugar das bibliotecas `cgi-lib.pl` e `cookie.lib` compatíveis com o *Perl 4*. Há também módulos `HTTP::*`, `HTTP::Response::*` e `HTTP::Request::*`. Os módulos são bastante extensos e possuem muitos métodos e formas de utilização. Em vez deles, preferimos apresentar o módulo `CGI_Lite.pm`. Ele contém as funções básicas presentes na biblioteca `cgi-lib.pl` (*Perl 4*) e ainda recursos para decodificar dados no formato `multipart/form-data` (resultantes de formulários do tipo *file-upload*). Ele não faz parte da distribuição original e não tem todos os recursos dos módulos CGI nativos mas é muito mais fácil de usar. Para instalá-lo, basta copiar o arquivo `CGI-Lite.pm` para o diretório `perl/lib/` da sua instalação. CGI-Lite pode ser encontrado em qualquer repositório CPAN (*Comprehensive Perl Archive Network*) na Internet.

Exemplo de uso

O programa abaixo decodifica todos os dados recebidos pelo formulário e os imprime em um arquivo de texto devolvido ao browser:

```
use CGI_Lite;

$cgi = new CGI_Lite();
$cgi->parse_form_data();

print "Content-type: text/plain","\n\n";
```

```
$cgi->print_form_data();
```

O método `parse_form_data` decodifica os dados de entrada e os retorna em um vetor associativo. Cada valor, do par nome-valor, pode ser recuperado da forma `$vetor{'nome'}`. O programa acima usou `print_form_data` para imprimir todos os valores. O programa abaixo faz o mesmo usando o vetor `%dados`:

```
$cgi = new CGI_Lite();
%dados = $cgi->parse_form_data();

print "Content-type: text/plain","\n\n";

foreach $key (keys %dados) {
   print $key, " = ", $dados{$key}, "\n";
}
```

Exemplo com File Upload [GUND96]

Formulários que usam o elemento HTML `<input type="file">` permitem que arquivos sejam enviados ao servidor pelo cliente. Os dados recebidos precisam ser separados e decodificados. O processo é simples com `CGI-Lite`.

```
#!c:\perl\bin\perl.exe

use CGI_Lite;
$cgi = new CGI_Lite();
print "Content-type: text/plain","\n\n";
$cgi->set_directory("c:\lixo") || die "Diretorio nao existe!\n";
$cgi->set_platform("DOS");
$cgi->set_file_type("handle");

%dados = $cgi->parse_form_data();
$email = $dados{'email'};
$ARQ   = $dados{'arquivo'};

print "Eis o arquivo que você nos enviou: \n";
print "-----------------------------------\n";
if (-T $arq) {
   while ($linha = <$ARQ>) {
       print $linha;
   }
   close $ARQ;
} else {
   print "Erro: você não enviou um arquivo de texto!";
}
print "\n-----------------------------------\n";
```

O arquivo HTML com um formulário que serve de interface ao programa acima pode ser o seguinte:

```
(...) <body>
<h1>Formulário de Upload</h1>
<form action="fup.pl" method="POST" enctype="multipart/form-data">
<p>Digite seu email <input type=text name=email size=15><br>
Arquivo de texto<br><input type=file name=arquivo><br>
<button type=submit>Enviar Arquivo</button></p>
</form>
</body> (...)
```

6.4. Módulo ODBC para Windows

Há vários módulos *Perl* para acesso a bancos de dados no *Unix*. No *Windows*, é preciso utilizar o *ActivePerl* (`www.activestate.com`) ou versão compatível, que possui alguns módulos nativos. Para acesso via ODBC existe um módulo em `http://www.roth.net` (compatível com o *ActivePerl*). Nesse site, pode-se baixar todo o módulo ODBC para instalação ou apenas os arquivos já compilados e colocá-los nos lugares adequados (mais fácil).

Os arquivos do pacote são dois: `ODBC.PM` e `ODBC.DLL`. O primeiro deve ser copiado para `\lib\Win32` da instalação *ActivePerl* (ex: *C*`:\perl\lib\Win32`). O arquivo `ODBC.PLL` deve ser copiado para `\lib\Auto\Win32\ODBC\`. Depois dessa instalação, o módulo já pode ser usado.

Como usar o módulo Win32::ODBC

Seu código deverá ter a seguinte linha:

```
use Win32::ODBC;
```

Em seguida, abra uma conexão na sua fonte de dados (cria objeto `$dados`):

```
$dados = new Win32::ODBC("DataSourceName");
```

Você pode agora enviar declarações SQL à vontade através do objeto que foi obtido (veja métodos abaixo). Quando terminar de usar o banco, chame o método `Close()`:

```
$dados->Close();
```

Principais métodos

Catalog *qualifier, owner, name, type*

Recupera o catálogo do objeto ODBC atual. Retorna um vetor de quatro elementos: (*Qualificador*, *Proprietário*, *Nome*, *Tipo*). Todos os nomes de campo usam caixa-alta. Exemplo:

```
($qualifier, $owner, $name, $type) = $db->Catalog("", "", "%", "'TA-
BLE'");
```

Close

Fecha a conexão.

Data

Data *list*

Recupera os dados de um cursor previamente carregado (como resultado de uma declaração SQL). Como escalar, retorna todos os campos concatenados. Como vetor, retorna cada campo em um elemento de vetor.
Exemplo:

```
$db->Sql("SELECT f1, f2, f3 FROM foo");
$db->FetchRow();
($f1, $f2) = $db->Data("f1", "f2");
```

ou

```
$db->Sql("SELECT * FROM foo");
$db->FetchRow();
@values = $db->Data;
```

DataHash

DataHash *list*

Recupera os dados de um cursor previamente carregado (como resultado de uma declaração SQL). Retorna uma tabela associativa (nome-valor) onde o nome do campo é a chave para recuperar o valor. Exemplo:

```
$db->Sql("SELECT f1, f2, f3 FROM foo");
$db->FetchRow();
%hash = $db->DataHash("f1", "f2");
print $hash{f1};
```

ou

```
$db->Sql("SELECT * FROM foo");
$db->FetchRow();
%hash = $db->DataHash;
foreach $key (sort(keys %hash)) {
   print $key, '=', $hash{$key}, "\n";
}
```

Error

Retorna o último erro na forma de um vetor ou um *string*. *Exemplo*:

```
die $db->Error(), qq(\n);
($ErrNum, $ErrText, $ErrConn) = $db->Error();
```

FetchRow

Busca o próximo registro (linha da coluna) da última declaração SQL. Para recuperar os dados, é preciso seguir um `FetchRow` por `Data` ou `DataHash`. Retorna `undef` se não houver mais linhas para ler. *Exemplo*:

```
$db->Sql("SELECT * FROM foo");
$db->FetchRow() || die qq(Fetch error: ), $db->Error(), qq(\n);
$f1 = $db->Data("f1");
```

TableList

TableList *qualifier, owner, name, type*

Recupera uma lista de nomes de tabela da conexão ODBC atual usando `Catalog`. *Exemplo*:

```
@tables = $db->TableList;
```

CGI para acesso via ODBC

O exemplo abaixo mostra um acesso simples a um banco de dados *Access*. Para que funcione é preciso que o módulo ODBC esteja instalado (seção anterior) e que haja uma fonte de dados ODBC (*Data Source*) no sistema com o nome "mdbdados1" que aponte para a base `anuncios.mdb`[3], distribuída com esta apostila. O programa apenas lista o conteúdo da base.

```
#!c:\perl\bin\perl.exe

use Win32::ODBC;

print "Content-type: text/html\n\n";
&buscaTudo;
exit(0);

sub buscaTudo
{
      $sql = "SELECT * FROM anuncios;";
      $dsn = "mdbdados1"; # este é o nome da fonte de dados do sistema
      $bd = new Win32::ODBC($dsn);
      $bd->Sql($sql);
      print "<table border=1>";
      while ($bd->FetchRow())
      {
            @registros = $bd->Data("numero", "data", "texto", "autor");
            print "<tr valign=top>";
            print "<td>$registros[0]</td>";
            print "<td>$registros[1]</td>";
            print "<td><pre>$registros[2]</pre></td>";
            print "<td>$registros[3]</td>";
            print "</tr>";
      }
      print "</table>";
      $bd->Close();
}
```

Veja outros exemplos, com acesso completo, atualização, inserção, remoção e busca nesta base, no disquete que acompanha esta apostila.

[3] O banco de dados anuncios.mdb contém apenas uma tabela chamada 'anuncios'. As colunas são 'numero' (INT), 'data' (CHAR), 'texto' (CHAR) e 'autor' (CHAR).

6.5. Módulo gráfico GD

O módulo gráfico GD é utilizado para gerar imagens PNG (ou GIF, nas versões mais antigas). Para isto dispõe de uma biblioteca de métodos e classes para a manipulação gráfica. Para instalar o GD é preciso baixar o módulo em `www.boutell.com`, compilá-lo e instalá-lo.

No *Unix*, a instalação geralmente não apresenta problemas e é simples, pois o roteiro de instalação é bastante eficiente.

No *Windows*, a compilação exige o *GNU C Compiler*, o que não é muito comum entre os usuários desse sistema. A melhor solução, neste caso, é usar o aplicativo *Perl Package Manager* (*PPM*) do *ActivePerl*. Estando conectado à Internet, rode *PPM* e, quando o prompt "`PPM>`" aparecer, digite "`install GD`" e aguarde uns 5 minutos (enquanto ele faz download dos arquivos). Depois de instalado, rode os programas de teste. Você pode usar a biblioteca incluindo no seu programa a declaração:

```
use GD;
```

Depois, crie um objeto `Image`, `Font` ou `Polygon` e chame seus métodos através dele.

Objeto Image

`GD::`**`Image`**

É uma classe que permite a criação de um objeto através do qual pode-se manipular os dados de uma imagem e chamar métodos de desenho e transformação. O objeto é criado através do construtor (método de classe) `new`:

```
$imagem = new GD::Image(50, 50);
$imagem->rectangle(0,0,40,40, $im->colorAllocate(0,0,255));
```

Veja um exemplo de criação e exibição de uma imagem (*Manual do GD*):

```
#!/usr/local/bin/perl

use GD;

# create a new image
$im = new GD::Image(100,100);

# allocate some colors
$white = $im->colorAllocate(255,255,255);
$black = $im->colorAllocate(0,0,0);
$red = $im->colorAllocate(255,0,0);
$blue = $im->colorAllocate(0,0,255);

# make the background transparent and interlaced
$im->transparent($white);
$im->interlaced('true');

# put a black frame around the picture
$im->rectangle(0,0,99,99,$black);

# Draw a blue oval
$im->arc(50,50,95,75,0,360,$blue);
```

```
# And fill it with red
$im->fill(50,50,$red);

# Convert the image to PNG and print it on standard output
print $im->png;
```

Objeto Polygon

GD::**Polygon**

Classe usada para criar um polígono. Os métodos que podem ser chamados a partir do objeto permitem acrescentar e remover vértices e realizar outras transformações.

```
$poly = new GD::Polygon;
$poly->addPt(50,0);
$poly->addPt(99,99);
$poly->addPt(0,99);
```

Objeto Font

GD::**Font**

Classe usada para criar novas fontes.

Construtor newFromGif (método de classe)

Este construtor cria uma imagem (manipulável dentro de um programa) através de uma imagem GIF existente, passada através de um descritor de arquivo previamente aberto para um arquivo GIF válido. Em caso de sucesso, o construtor retorna a imagem como um objeto. Em caso de falha, retorna `undef`.

```
open (ANTAS,"duasantas.gif") || die "Impossivel abrir arquivo";
$im = newFromGif GD::Image(ANTAS) || die "Formato incorreto";
close GIF;
```

Principais métodos do objeto GD::Image

colorAllocate

GD::Image::**colorAllocate**(*r*, *g*, *b*)

Este método aloca uma cor de acordo com seus componentes de luz (vermelha, verde e azul) que podem ter valores que variam entre 0 e 255. Se uma cor não for alocada, este método retorna –1. A primeira cor alocada é utilizada como cor de fundo. *Exemplo*:

```
$white = $im->colorAllocate(255,255,255); # cor de fundo
$black = $im->colorAllocate(0,0,0);
$magenta = $im->colorAllocate(255,0,255);
```

line

GD::Image::**line**(*x1*, *y1*, *x2*, *y2*, *cor*)

Desenha uma linha de (*x1, y1*) a (*x2, y2*) na cor especificada. Pode-se usar uma cor real, previamente alocada, ou uma das cores especiais `gdBrushed`, `gdStyled` e `gdStyledBrushed`. *Exemplos*:

```
$im->line(0,0,150,150,gdBrushed);
$im->line(0,150,150,0,$magenta);
$im->line(0,75,150,75, $im->colorAllocate(255,0,0));
```

rectangle

`GD::Image::`**`rectangle`**(*x1, y1, x2, y2, cor*)

Desenha um retângulo na cor especificada. Os pontos (*x1, y1*) e (*x2, y2*) correspondem aos cantos superior esquerdo e inferior direito, respectivamente. Pode-se ainda usar as cores especiais `gdBrushed`, `gdStyled` e `gdStyledBrushed`. *Exemplo*:

```
$im->rectangle(10,10,100,100,$magenta);
```

filledRectangle

`GD::Image::`**`filledRectangle`**(*x1, y1, x2, y2, cor*)

Preenche um retângulo na cor especificada. Os pontos (*x1, y1*) e (*x2, y2*) correspondem aos cantos superior esquerdo e inferior direito, respectivamente. Pode-se usar uma cor real (previamente alocada) ou com o padrão definido por uma imagem (previamente carregada usando `newFromGif`) definida com o método `setTile()`. O padrão (*tile*) atual pode ser atribuído ao desenho com a "cor" reservada `gdTiled`. Exemplo:

```
open(GIF,"tijolos.gif") || die;
$tile = newFromGif GD::Image(GIF);
$myImage->setTile($tile);
$myImage->filledRectangle(10,10,150,200,gdTiled);
```

polygon

`GD::Image::`**`polygon`**(*polígono, cor*)

Desenha um polígono na cor especificada. É preciso, antes de desenhá-lo, criar o polígono usando a classe `GD::Polygon`. Um polígono deve ter pelo menos três vértices. Se o último vértice não fechar o polígono, o método o fechará automaticamente antes de desenhá-lo. Pode-se usar cores reais ou as cores especiais `gdBrushed`, `gdStyled` e `gdStyledBrushed` para desenhar o polígono (não preenchido). Exemplo:

```
$poly = new GD::Polygon;
$poly->addPt(50,0);
$poly->addPt(99,99);
$poly->addPt(0,99);
$im->polygon($poly,$blue);
```

filledPolygon

`GD::Image::`**`filledPolygon`**(*polígono, cor*)

Desenha e preenche o polígono com a cor especificada. Pode-se usar as cores comuns ou a cor especial `gdTiled` (padrão baseado em imagem previamente importada). *Exemplo*:

```
$poly = new GD::Polygon;
$poly->addPt(50,0);
$poly->addPt(99,99);
$poly->addPt(0,99);
$im->filledPolygon($poly, $magenta);
```

arc

GD::Image::**arc**(*cx*, *cy*, *largura*, *altura*, *início*, *fim*, *cor*)

Desenha arcos e elipses. O centro é especificado em *cx*, *cy*) e (*largura*, *altura*) especificam a altura e largura. O *início* e o *fim* são definidos em graus de 0 a 360. Zero é a parte mais alta do elipse. Noventa (90) é a parte mais à direita. Para desenhar círculos, use início em 0, fim em 360 e valores iguais para altura e largura. Pode-se usar cores reais ou as cores especiais `gdBrushed`, `gdStyled` e `gdStyledBrushed`. *Exemplo*:

```
$im->arc(100,100,50,50,0,180,$blue);
```

fill

GD::Image::**fill**(*x*, *y*, *color*)

Preenche uma região na cor especificada. Funciona como a ferramenta "balde de tinta" nos aplicativos de desenho. A pintura começa na origem (*x, y*) e pára logo que encontra um pixel de outra cor. Pode-se usar cores normais ou `gdTiled`. *Exemplo*:

```
$im->rectangle(10,10,100,100,$black);
$im->fill(50,50,$blue);
```

fillToBorder

GD::Image::**fillToBorder**(*x*, *y*, *cor_da_borda*, *cor_do_preenchimento*)

Como `fill`, este método preenche uma área com a cor especificada mas permite que se defina outra cor para a borda. A borda só pode usar uma cor normal (não pode usar as cores especiais). O preenchimento pode ser uma cor normal ou `gdTiled`. *Exemplo*:

```
$im->rectangle(10,10,100,100,$red);
$im->fillToBorder(50,50,$black,$blue); # borda será azul
```

string

GD::Image::**string**(*font*, *x*, *y*, *string*, *cor*)

Este método desenha uma *string* na posição (*x,y*) utilizando fonte e cor especificadas. Há quatro fontes diferentes que podem ser usadas: `gdSmallFont`, `gdMediumBoldFont`, `gdTinyFont` e `gdLargeFont`. *Exemplo*:

```
$im->string(gdSmallFont,2,10,"Imagem vale 1000 palavras", $magenta);
```

png (ou gif)

GD::Image::**gif**

Retorna o código (texto) da imagem em formato PNG (*Portable Network Graphics*) ou GIF (*Graphics Image Format*). Apenas um dos formatos está disponível dependendo da versão do GD utilizada. As versões mais novas suportam apenas PNG uma vez que o formato GIF agora é proprietário.

```
$imagemPNG = $im->png;
print $imagemPNG;
```

Consulte manual do GD para outros métodos e métodos dos objetos `Polygon` e `Font`.

7. Referências

1. Randal Schwartz. *Learning Perl*. O'Reilly and Associates, 1993
2. Larry Wall. *Programming Perl*. O'Reilly and Associates, 1992
3. Shishir Gundavaram. *CGI Programming for the World Wide Web*. O'Reilly and Associates, 1996
4. ActiveState. *ActivePerl 522 User´s Manual*. www.activestate.com.
5. *Perl 5.004 Manual Pages*. www.perl.org
6. *Perl 5.004 Frequently Asked Questions*. www.perl.org
7. *GD Graphics Library*. www.boutell.com
8. Roth. *Windows ODBC Module*. www.roth.net
9. Spainhour and Quercia. *WebMaster in a NutShell*. O'Reilly and Associates, 1997